МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Программной инженерии
Специальность 6-05-0612-01 Программное инженерия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка компилятора RIA-2025»

Выполнил студент Рубцов Иван Александрович
(Ф.И.О. студента)

Руководитель проекта преп.-стажер Авдеева Вера Дмитриевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты преп.-стажер Авдеева Вера Дмитриевна
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Минск 2025

Содержание

Введение

Целью курсовой работы является разработка транслятора для языка программирования RIA-2025. Главной задачей транслятора является, перобразование программы, написанной на языке программирования RIA-2025, в программу, которая будет понятна компьютеру. В данном курсовом проекте трансляция будет осуществляться в код на языке Assembler.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;
– разбратка структуры транслятора;
– разработка лексического анализатора;
– разработка синтаксического анализатора;
– разработка семантического анализатора;
– обработка выражений;
– генерация кода на язык Assebler;
– тестирование транслятора.

Язык программирования RIA-2025 предназначен для выполнения простейших арифметический действий, операций над строками и числами.

1 Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования RIA-2025 является процедурным, строго типизированным, не объектно-ориентированным, компилируемым языком.

Процедурный язык программирования – язык высокого уровня, в котором используется метод разбиения программ на отдельные связанные между собой модули – подпрограммы.

Строго типизированный язык программирования – язык, в котором переменные привязаны к конкретным типам данных. Язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования.

Объектно-ориентированный язык программирования – язык, построенный на принципах объектно-ориентированного программирования. В основе концепции объектно-ориентированного программирования лежит понятие объекта – некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Компилируемый язык программирования – язык программирования, исходный код которого преобразуется компилятором в исходный код на другом языке программирования.

1.2 Определение алфавита языка программирования

В основе алфавита RIA-2025 лежит таблица символов Windows-1251. Исходный код RIA-2025 может содержать символы латинского и русского алфавита, цифры десятичной системы счисления от 0 до 9.

Таблица 1.1

Название подгруппы	Символы подгруппы
Символы латинского алфавита	[a-z] && [A-Z] && [а-я]
Специальные и числовые символы	[‘ - >] && [{ - }] && [[-]] && символ “ “ (пробел) :

В языке RIA-2025 символы разбиты на подгруппы, включая символы латинского и кириллического алфавита, обозначаемые диапазонами [a-z], [A-Z] и [а-я]. Также выделена подгруппа специальных и числовых символов, которая включает различные знаки препинания, скобки и пробелы для корректного построения программных конструкций.

1.3 Применяемые сепараторы

Язык RIA-2025 разрешает использовать сепараторы для написания исходного кода, представленные в таблице 1.2. Сепараторы помогают отделять инструкции, блоки кода и элементы выражений, обеспечивая правильное восприятие программы транслятором. Они делают код более

читаемым для программиста и упрощают его отладку. Правильное использование сепараторов является важным условием для корректной работы компилятора и предотвращения синтаксических ошибок.

Таблица 1.2 Символы-сепараторы языка RIA-2025

Символ(ы)	Назначение
‘пробел’	Разделитель цепочек. Допускается везде кроме названий идентификаторов, ключевых слов и инициализации строки
{ … }	Блок функции или условной конструкции
(…)	Блок фактических или формальных параметров функции, а также приоритет арифметических операций
,	Разделитель параметров функций
+ - */	Арифметические операции
> < == !=	Логические операции (операции сравнения: больше, меньше, проверка на равенство, на неравенство), используемые в условных конструкциях.
;	Разделитель программных конструкций
=	Оператор присваивания

Символы-сепараторы языка RIA-2025 служат для структурирования исходного кода, разграничения конструкций и обозначения операций, обеспечивая корректное разбиение программы на элементы.

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования RIA-2025 используется кодировка ASCII, представленная на рисунке 1.1.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рисунок 1.1 -Алфавит входных символов

Исходный код на языке RIA-2025 создаётся с использованием кодировки ASCII, что гарантирует стандартное представление всех символов. Каждый символ, включая буквы, цифры и служебные знаки, имеет

определённое числовое значение в этой кодировке. Это обеспечивает правильное считывание и обработку программного текста транслятором на всех этапах компиляции.

1.5 Типы данных

В языке RIA-2025 есть четыре типа данных: беззнаковый целый, логический, целочисленный и строковый. Пользовательские типы данных не поддерживаются. Описание типов данных, предусмотренных в данном языке, представлено в таблице 1.3. Эти типы данных являются основой для построения всех программ, написанных на данном языке. Беззнаковый целый тип предназначен для работы с натуральными числами и различными счётчиками. Логический тип используется для представления двух логических значений — истинно и ложно. Ограниченное количество типов данных позволяет упростить процесс разработки и снизить вероятность ошибок. Благодаря этому язык ориентирован на учебные и исследовательские задачи, где важна предсказуемость и простота поведения программ.

Таблица 1.3 – Типы данных языка RIA-2025

Тип данных	Описание типа данных
1	2
Логический тип данных **bool**	Фундаментальный тип данных. Используется для представления логических (истинностных) значений — истина (true) или ложь (false). Каждое логическое значение занимает 1 байт памяти. Операции над данными логического типа: Логические переменные могут участвовать в выражениях, операциях сравнения и логических операциях. Поддерживаемые операции: = (бинарный) – оператор присваивания. ! – логическое отрицание (NOT). && – логическое И (AND). || – логическое ИЛИ (OR). Операции сравнения возвращают значение типа bool.
Беззнаковый целочисленный тип данных **unsigned integer**	Фундаментальный тип данных. Поддерживаемые операции: + (бинарный) – оператор сложения; - (бинарный) – оператор вычитания; * (бинарный) – оператор умножения; / (бинарный) – оператор деления; = (бинарный) – оператор присваивания. В качестве условия условного оператора поддерживаются следующие логические операции: > (бинарный) – оператор «больше»; < (бинарный) – оператор «меньше»; == (бинарный) – оператор проверки на равенство; != (бинарный) – оператор проверки на неравенство.

Продолжение таблицы 1.3

1	2
Целочисленный тип данных **integer**	Хранит целочисленный тип данных. Используется для представления знаковых числовых значений в программе.
Строковый тип данных **string**	Хранит указатель на начало строки. Представляет последовательность символов, оканчивающуюся нулевым символом.

Логический тип данных bool представляет значения истинности и используется в условиях, ветвлениях и логических выражениях. Он занимает 1 байт памяти и по умолчанию инициализируется значением false. Переменные типа bool могут участвовать во всех логических операциях, включая отрицание, логическое И и логическое ИЛИ. Результаты операций сравнения также имеют тип bool, что позволяет удобно использовать их в управляющих конструкциях языка.

1.6 Преобразование типов данных

В языке программирования RIA-2025 преобразование типов данных не поддерживается, т.к. язык является строго типизированным.

1.7 Идентификаторы

В языке RIA-2025 идентификаторы должны быть составлены только из символов нижнего регистра английского алфавита. Типы идентификаторов: имя переменной или функции, параметр, имя стандартной функции. Идентификатор составляется из букв латинского алфавита от 1 до 12 символов, без пробелов. Максимальная длина идентификатора равна 12 символам. Идентификатор не может совпадать с ключевыми словами.

<идентификатор> ::= <буква> |<буква><идентификатор>

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

1.8 Литералы

С помощью литералов осуществляется инициализация переменных, позволяющая задать их начальные значения непосредственно в исходном коде. В языке RIA-2025 предусмотрено два основных вида литералов: литералы целого типа и строковые литералы, которые используются для задания чисел и строковых данных соответственно. Литералы целого типа могут задаваться в двух системах счисления: в привычной десятичной форме, а также в шестнадцатеричной, что упрощает работу с низкоуровневыми значениями и адресами. Строковые литералы заключаются в кавычки и позволяют задавать последовательности символов, которые будут храниться в виде строк. Подробное описание всех видов литералов, их структуры, правил записи и примеров использования представлено в таблице 1.4, что обеспечивает удобство при разработке программ на языке RIA-2025.

Таблица 1.4 – Описание литералов

Тип литерала	Пояснение	Пример
Строковый	Набор символов алфавита языка, заключенных в двойные кавычки.	declare string str; str = 'Hello world'; 'Hello world' – строковый литерал.
Логический	Может принимать только два значения: true (истина) или false (ложь).	declare bool flag; flag = true;

Строковые литералы в языке RIA-2025 представляют собой набор символов, заключённых в двойные кавычки, и не могут содержать пробелы. Логические литералы могут принимать только два значения – true или false, и используются для задания логических переменных.

1.9 Объявление данных

В языке RIA-2025 объявление данных начинается с ключевого слова declare, указывается тип данных и имя идентификатора.

Пример: declare unsigned integer a, declare string b;

Область видимости: сверху вниз, параметры внутри функции, объявления внутри функции видны только внутри функции, объявления переменных за пределами функций и главной функции предусмотрены.

1.10 Инициализация данных

Инициализация переменной происходит после её объявления. Инициализация переменной в момент объявления запрещена.

Например: declare string word; word = “слово”; declare unsigned num; num = 5;

1.11 Инструкции языка

Все возможные инструкции языка программирования RIA-2025 представлены в общем виде в таблице 1.5.

Таблица 1.5 – Инструкции языка программирования RIA-2025

Инструкция	Запись на языке RIA-2025
1	2
Объявление переменной	declare <тип данных> <идентификатор>;
Точка входа	main { … }
Объявление внешней функции	function <тип данных> <идентификатор> (<тип данных> <идентификатор>, …) {…}
Инициализация переменной	<идентификатор> = <выражение>; Выражением может быть идентификатор, литерал, или вызов функции соответствующего типа.

Продолжение таблицы 1.5

1	2
Возврат значения из подпрограммы	return <идентификатор> | <литерал>;
Вывод данных	print (<идентификатор> | <литерал>);

Условный оператор	if (<условие> | (<идентификатор> | <литерал>)
	{
	…
	}
	Блок else не предусмотрен.

В языке RIA-2025 предусмотрены базовые инструкции, такие как объявление переменных, точка входа программы и определение внешних функций. Все операции инициализации, вывода данных и возврата значений выполняются с использованием простых и однозначных конструкций, что упрощает чтение и понимание кода. Условный оператор поддерживает только ветвление с одним условием, без блока else, что подчёркивает минималистичную структуру языка. Инструкции языка имеют строгий формат записи, поэтому каждая операция должна соответствовать установленному синтаксису. Благодаря этому программы на RIA-2025 становятся предсказуемыми и удобными для анализа.

1.12 Операции языка

В языке RIA-2025 предусмотрены следующие операции с данными. Приоритетность операций определяется с помощью (). Операции представлены в таблице 1.6. Дополнительно каждая операция имеет строго определённый тип входных аргументов, что позволяет избежать неоднозначностей при вычислениях. Корректное использование операций обеспечивает предсказуемое поведение выражений и упрощает последующую генерацию кода.

Таблица 1.6 Операции языка RIA-2025

Операция	Описание
+	Бинарный, суммирование
-	Бинарный, вычитание
*	Бинарный, умножение
/	Бинарный, деление
<	Бинарный, меньше
>	Бинарный, больше
==	Бинарный, равенство
!=	Бинарный, неравенство
%	Бинарный, получение остатка от деления

В языке RIA-2025 все перечисленные операции являются бинарными и выполняются между двумя операндами. Они включают арифметические операции (+, -, *, /, %) и операции сравнения (<, >, ==, !=, <=, >=), каждая из которых имеет своё определённое назначение.

1.13 Выражения и их вычисления

Вычисление выражений – одна из важнейших задач языков программирования. Всякое выражение составляется согласно следующим правилам:

– допускается использовать скобки для смены приоритета операций;

– выражение записывается в строку без переносов;

– использование двух подряд идущих операторов не допускается;

Перед генерацией кода каждое выражение приводится к записи в польской записи для удобства дальнейшего вычисления выражения на языке ассемблера.

1.14 Конструкции языка

В языке программирования RIA-2025 предусмотрена одна главная функция, которая служит точкой входа для выполнения программы, а также внешние функции, позволяющие структурировать и дополнять её логику. Главная функция определяет основное поведение программы, тогда как внешние функции используются для реализации отдельных действий и повторяющихся операций. Такая организация кода делает программы более понятными и удобными для сопровождения. Программные конструкции, доступные в языке, представлены в таблице 1.7 и позволяют разработчику использовать базовые элементы для описания алгоритмов и взаимодействия между частями программы.

Таблица 1.7 Программные конструкции RIA-2025

Внешняя функция	function идентификатор (<тип данных> <идентификатор>, …)
	{
	…
	return <идентификатор / литерал>;
	}
	Область видимости сверху вниз. Все переменные являются локальными.

Главная функция	main
	{
	…
	}
	Область видимости сверху вниз. Переменные являются локальными.

Условная конструкция	if(<идентификатор/литерал> <знак логической операции> <идентификатор/литерал>)
	{
	…}

Внешние функции в языке RIA-2025 позволяют выполнять отдельные логические блоки программы и возвращают значение с помощью оператора return. Как внешние, так и главная функция имеют область видимости сверху вниз, при этом все объявленные внутри переменные являются локальными. Условная конструкция if использует логическое сравнение идентификаторов или литералов и выполняет код внутри блока только при выполнении указанного условия.

1.15 Область видимости идентификаторов

Область видимости идентификаторов в языке RIA-2025 – локальная внутри программных блоков функций.

Сверху вниз, параметры внутри функции, объявления внутри функции видны только внутри функции, объявления за пределами функций и главной функции допускаются.

1.16 Семантические проверки

Основные семантические правила языка RIA-2025, проверяемые на различных этапах работы транслятора, представлены в таблице 1.8 и служат основой для корректного анализа программ. Эти правила определяют, какие конструкции считаются допустимыми, как должны взаимодействовать элементы языка и какие ошибки необходимо обнаруживать ещё до выполнения программы. Часть семантических проверок выполняется уже на этапе лексического анализа, где контролируется корректность структуры входной последовательности символов и соответствие идентификаторов установленным правилам. Благодаря многоэтапной проверке удаётся своевременно выявлять типичные ошибки программиста и обеспечивать надёжность процесса трансляции.

Таблица 1.8 – Семантические правила

Номер	Правило
1	Должна присутствовать точка входа main и только одна
2	Идентификаторы должны быть объявлены до инициализации и использования
3	Не должно быть объявлений идентификаторов с одинаковыми именами в одном и том же блоке кода
4	Присваивать значение идентификатору можно только соответствующего типа
7	Вызов функции обязует использование скобок после ее названия с передачей параметров соответствующих типов или без них
8	Тип возвращаемого функцией значения должен соответствовать типу функции
9	Деление на ноль запрещено
10	Проводить арифметические операции со строковым типом данных запрещено
11	Превышение размеров строковых и целочисленных литералов
12	Выражение должно быть условным

Перечень семантических правил языка RIA-2025 определяет корректность структуры программы и предотвращает возникновение ошибок при выполнении. Эти правила контролируют правильность объявления идентификаторов, типизацию переменных и корректность вызовов функций. Также они включают ограничения на арифметические операции, проверку допустимых значений литералов и требование, чтобы выражения, используемые в условиях, были истинностными.ф

1.17 Распределение оперативной памяти на этапе выполнения

Транслированный код использует две области памяти. В сегмент констант заносятся все литералы. Локальная область видимости в исходном коде определяется правилами именования идентификаторов и их префиксами, что обеспечивает локальность на уровне исходного кода, хотя в оттранслированном ассемблерном коде переменные имеют глобальную область видимости.

1.18 Стандартная библиотека и её состав

В RIA-2025 присутствует стандартная библиотека. Возможные функции стандартной библиотеки описаны в таблице 1.9.

Таблица 1.9 Стандартная библиотека

Функция	Описание
tostring(идентификатор или литерал);	Преобразует значение типа int в строку (string).
strlen (идентификатор или литерал);	Вычисляет длину строки (количество символов до нулевого терминатора '\0'). Применима для идентификаторов типа string и строковых литералов.

Стандартная библиотека написана на языке C++, подключается к транслированному коду на этапе генерации кода. Вызовы стандартных функций доступны там же, где и вызов пользовательских функций. Также в стандартной конечному пользователю.

1.19 Ввод и вывод данных

Вывод данных осуществляется с помощью оператора cout. Допускается использование оператора cout с литералами и идентификаторами.

Функции, управляющие выводом данных, реализованы на языке C++ и вызываются из транслированного кода, конечному пользователю недоступны. Пользовательская команда cout в транслированном коде будут заменена вызовом нужных библиотечных функций. Библиотека, содержащая нужные процедуры, подключается на этапе генерации кода.

1.20 Точка входа

В языке RIA-2025 каждая программа должна содержать главную функцию (точку входа) main. Функция точки входа представлена в таблице 1.10.

Таблица 1.10 – Точка входа

Конструкция	Реализация

Главная функция
(точка входа)
	main
	{
	/ программный блок /
	}

Главная функция main является обязательной конструкцией в языке RIA-2025 и служит начальной точкой выполнения программы. Она оформляется в виде блока кода в фигурных скобках, внутри которых размещаются все инструкции, составляющие основную логику программы.

1.21 Препроцессор

Препроцессор, принимающий и выдающий некоторые данные на вход транслятору, в языке RIA-2025 отсутствует.

1.22 Соглашения о вызовах

Соглашение о вызовах – это правила передачи управления от вызывающего к вызываемому коду, определяющие способы передачи параметров и результата вычислений, возврат в точку вызова.

В языке RIA-2025 вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память освобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

1.23 Объектный код

Язык программирования RIA-2025 транслируется в язык ассемблера.

1.24 Классификация сообщений транслятора

Генерируемые транслятором сообщения определяют степень его информативности, то есть сообщения транслятора должны давать максимально полную информацию о допущенной пользователем ошибке при написании программы. Классификация ошибок транслятора приведены в таблице 1.11. Дополнительная детализация сообщений позволяет пользователю быстрее определить причину возникшей проблемы и локализовать её в исходном коде. Такой подход повышает удобство разработки и снижает вероятность повторного возникновения тех же ошибок.

Таблица 1.11 Классификация ошибок

Номера ошибок	Характеристика
0 – 99	Системные ошибки
100 – 299	Ошибки лексического анализа
300 – 399	Ошибки семантического анализа
600 – 699	Ошибки синтаксического анализа
400-499, 700-999	Зарезервированные коды ошибок

В языке RIA-2025 коды ошибок разделены на диапазоны, каждый из которых соответствует определённому типу ошибок, включая системные, лексические, семантические и синтаксические. Диапазоны 400–499 и 700–999 зарезервированы для будущего использования или специальных случаев.

1.25 Контрольный пример

Контрольный пример представлен в приложении А.

2 Структура транслятора

2.1 Компоненты транслятора, их назначение и принципы взаимодействия

В языке RIA-2025 исходный код транслируется в язык Assembler. Транслятор языка разделён на отдельные части, которые взаимодействуют между собой и выполняют отведённые им функции, которые представлены в пункте 2.1. Для того чтобы получить ассемблерный код, используется выходные данные работы лексического анализатора, а именно таблица лексем и таблица идентификаторов. Для указания выходных файлов используются входные параметры транслятора, которые описаны в таблице 2.1. Структура транслятора языка RIA-2025 приведена на рисунке 1.

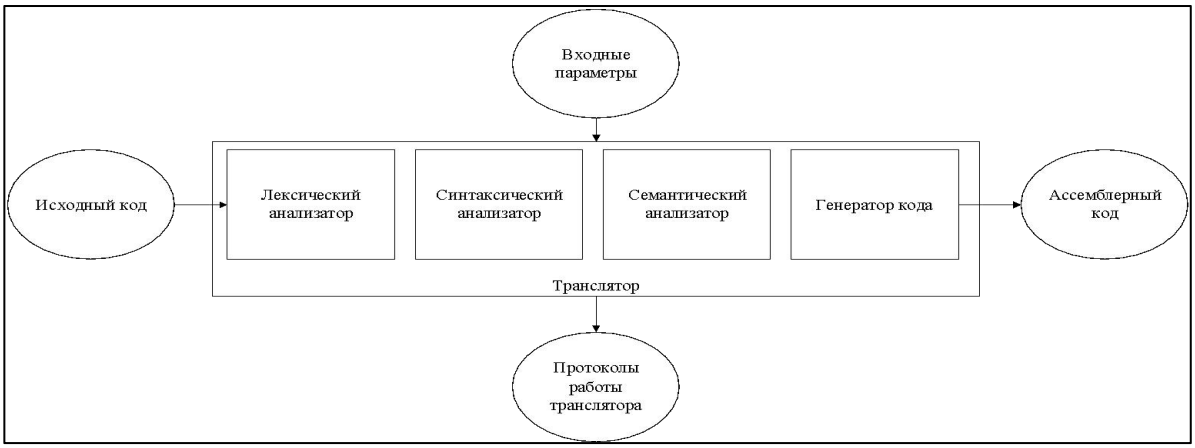


Рисунок 2.1 Структура транслятора языка программирования RIA-2025

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся последовательность символов входного языка. Он производит предварительный разбор текста, преобразующий единый массив текстовых символов в отдельные слова. Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация. Таблица лексем (ТЛ) и таблица идентификаторов (ТИ) являются входом для следующей фазы компилятора – синтаксического анализа (разбора, парсера).

Цели лексического анализатора:

− убрать все лишние пробелы;

− выполнить распознавание лексем;

− построить таблицу лексем и таблицу идентификаторов;

− при неуспешном распознавании или обнаружении некоторых ошибок во входном тексте выдать сообщение об ошибке.

Синтаксический анализатор – часть компилятора, выполняющая синтаксический анализ, то есть проверку исходного кода на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией является дерево разбора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть проверку исходного кода на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию ассемблерного кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

2.2 Перечень входных параметров транслятора

Для формирования файлов с результатами работы лексического, синтаксического и семантического анализаторов используются входные параметры транслятора, которые приведены в таблице 2.1.

Таблица 2.1 Входные параметры транслятора языка RIA-2025

Входной параметр	Описание параметра	Значение по умолчанию
-in:<путь к in-файлу>	Файл с исходным кодом на языке RIA-2025, имеющий расширение .txt	Не предусмотрено
-log:<путь к log-файлу>	Файл журнала для вывода протоколов работы программы.	Значение по умолчанию: <имя in-файла>.log

Входной параметр -in задаёт путь к файлу с исходным кодом на языке RIA-2025 и является обязательным для работы программы. Параметр -log указывает файл журнала для вывода протоколов работы, по умолчанию его имя совпадает с именем входного файла с расширением .log.

2.3 Протоколы, формируемые транслятором

В ходе работы программы формируются протоколы работы лексического, синтаксического и семантического анализаторов, которые содержат в себе перечень протоколов работы. В таблице 2.2 приведены протоколы, формируемые транслятором, и их содержимое. Каждый протокол фиксирует последовательность операций и выявленные ошибки на соответствующем этапе анализа. Это позволяет разработчику отслеживать корректность обработки исходного кода и выявлять проблемные участки. Протоколы служат важным инструментом для отладки и оптимизации программы, обеспечивая прозрачность работы транслятора.

Таблица 2.2 Протоколы, формируемые транслятором языка RIA-2025;

Формируемый протокол	Описание выходного протокола
Файл журнала, заданный параметром "-log:"	Файл с протоколом работы транслятора языка программирования RIA-2025. Содержит таблицу лексем и таблицу идентификаторов, протокол работы синтаксического анализатора и дерево разбора, полученные на этапе лексического и синтаксического анализа, а также результат работы алгоритма преобразования выражений к польской записи.

Формируемый протокол создаётся в специальном файле журнала, который задаётся параметром «-log:» при запуске транслятора. В этом файле содержится полный и подробный отчёт о работе всех этапов трансляции языка RIA-2025, что позволяет проследить процесс обработки программы от начала и до конца. Протокол включает таблицу лексем и таблицу идентификаторов, формируемые лексическим анализатором, а также детальное описание работы синтаксического анализатора.

Кроме того, в журнал заносится дерево разбора, позволяющее визуально представить структуру программы и проверить корректность её построения. В протокол также добавляется результат преобразования выражений в польскую запись, который необходим для дальнейших этапов трансляции и анализа. Такой расширенный отчёт облегчает поиск ошибок, делает процесс разработки более прозрачным и помогает пользователю точнее понять работу транслятора.

3 Разработка лексического анализатора

3.1 Структура лексического анализатора

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (сканером). На вход лексического анализатора подаётся исходный код входного языка. Лексический анализатор выделяет в этой последовательности простейшие конструкции языка, Лексический анализатор производит предварительный разбор текста, преобразующий единый массив текстовых символов в массив токенов.

Примеры лексических единиц: идентификаторы, числа, символы операций, служебные слова и т.д. Лексический анализатор преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами, для создания промежуточного представления исходной программы. Каждой лексеме сопоставляется ее тип и запись в таблице идентификаторов, в которой хранится дополнительная информация.

Функции лексического анализатора:

− удаление «пустых» символов и комментариев. Если «пустые» символы (пробелы, знаки табуляции и перехода на новую строку) и комментарии будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними (альтернативный способ, состоящий в модификации грамматики для включения «пустых» символов и комментариев в синтаксис, достаточно сложен для реализации);

− распознавание идентификаторов и ключевых слов;

− распознавание констант;

− распознавание разделителей и знаков операций.

Исходный код программы представлен в приложении А, структура лексического анализатора представлена на рисунке 3.1.

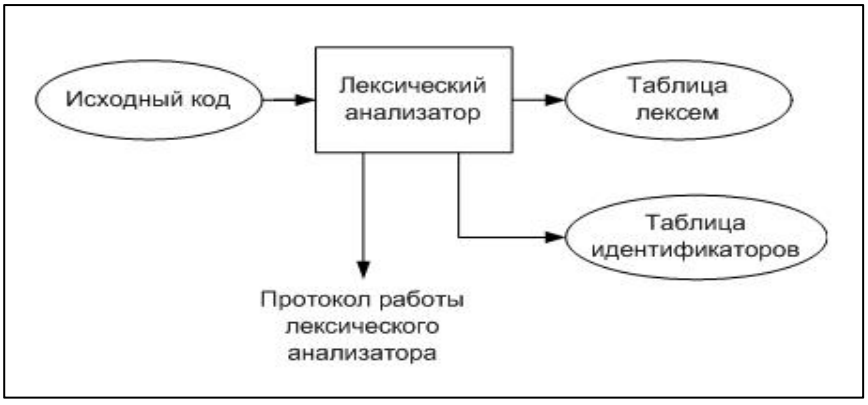


Рисунок 3.1 Структура лексического анализатора

Первая стадия работы компилятора — лексический анализ, который выполняет программа, называемая лексическим анализатором или сканером. Он преобразует исходный код в последовательность токенов, выделяя идентификаторы, числа, ключевые слова, операторы и разделители, и создаёт

промежуточное представление программы. Основные функции лексического анализатора включают удаление пустых символов и комментариев, распознавание идентификаторов, констант и знаков операций для последующей обработки синтаксическим анализатором.

3.2 Контроль входных символов

Для удобной работы с исходным кодом, при передаче его в лексический анализатор, все символы разделяются по категориям. Таблица входных символов представлена на рисунке 3.2, категории входных символов представлены в таблице 3.1.

```
#define IN_CODE_TABLE {\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::P, IN::N, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::P, IN::S, IN::Q, IN::S,  IN::T, IN::S, IN::S, IN::Q,  IN::S, IN::S, IN::S, IN::S,  IN::S, IN::S, IN::T, IN::S,\
                IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::S,  IN::S, IN::S, IN::S, IN::T,\
                IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,\
                IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::S,  IN::T, IN::S, IN::T, IN::T,\
                IN::F, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,\
                IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::T,  IN::T, IN::T, IN::T, IN::S,  IN::S, IN::S, IN::S, IN::T,\
                                                                                                                      \
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,\
                IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F,  IN::F, IN::F, IN::F, IN::F \
                                             }
```

Рисунок 3.2. – Таблица контроля входных символов

Для эффективного анализа исходного кода лексический анализатор распределяет все символы по определённым категориям. Информация о категории этих символов — в таблице 3.1.

Таблица 3.1 Соответствие символов и их значений в таблице

Значение в таблице входных символов	Символы
Разрешенный	T
Запрещенный	F
Игнорируемый	I

В таблице входных символов каждому символу присваивается значение, определяющее его допустимость: разрешённый, запрещённый или игнорируемый. Разрешённые символы обозначаются как T, запрещённые — F, а игнорируемые — I.

3.3 Удаление избыточных символов

Избыточными символами являются символы табуляции и пробелы. Избыточные символы удаляются на этапе разбиения исходного кода

на токены.

Описание алгоритма удаления избыточных символов:

– посимвольно считываем файл с исходным кодом программы;

– встреча пробела или знака табуляции является своего рода встречей символа-сепаратора;

– в отличие от других символов-сепараторов не записываем в очередь лексем эти символы, т.е. игнорируем.

3.4 Перечень ключевых слов

Лексический анализатор преобразует исходный текст, заменяя лексические единицы лексемами для создания промежуточного представления исходной программы. Соответствие токенов и лексем приведено в таблице 3.2.

Таблица 3.2 Соответствие токенов и сепараторов с лексемами

Токен	Лексема	Пояснение
Unsigned, integer, string, bool	t	Названия типов данных языка.
Идентификатор	i	Длина идентификатора – 12 символов.
Литерал	l	Литерал любого доступного типа.
function	f	Объявление функции.
return	r	Выход из функции.
main	m	Главная функция.
declare	d	Объявление переменной.
if	u	Условный оператор
do-while	d-w	Цикл
;	;	Разделение выражений.
,	,	Разделение параметров функций.
{	{	Начало блока/тела функции.
}	}	Закрытие блока/тела функции.
(	(	Передача параметров в функцию, приоритет операций, условие.
)	)	Закрытие блока для передачи параметров, приоритет операций, условия.
=	=	Знак присваивания.
+ - * /	v	Знаки операций.
> < == !=	s	Знаки логических операторов

Каждому выражению соответствует конечный автомат, по которому происходит разбор данного выражения. Для ключевых слов используется детерминированный конечный автомат, а для идентификаторов и литералов недетерминированный. На каждый автомат в массиве подаётся токен и с

помощью регулярного выражения, соответствующего данному графу переходов, происходит разбор. В случае успешного разбора выражения оно записывается в таблицу лексем. Если выражение является идентификатором или литералом, информация также заносится в таблицу идентификаторов. Структура конечного автомата и пример графа перехода конечного автомата находятся в листингах 3.3 и 3.4 соответственно.

```
namespace FST
{
     struct RELATION     // ребро:символ -> вершима графа
переходов КА
     {
          char symbol;   // символ перехода
          short nnode;   // номер смежной вершины
          RELATION(
               char c = 0x00, // символ перехода
               short ns = NULL     // новое состояние
          );
     };
     struct NODE // вершина графа переходов
     {
          short n_relation;   // количество инциндентных ребер
          RELATION* relations;     // инцидентные ребра
          NODE();
          NODE(
               short n,  // количество инциндентных ребер
               RELATION rel, ...   // список ребер
          );
};
     struct FST               // недетерменированный конечный
автомат
     {char* string;      // цепочка (строка, завершается 0x00)
          short position;          // текущая позиция в цепочке
          short nstates;      // количество состояний автомата
          NODE* nodes;        // граф переходов: [0] - начальное
состояние, [nstate-1] - конечне
          short* rstates;          // возможные состояния
автомата на данной позиции
          FST();
          FST(
               char* s,            // цепочка (строка,
завершается 0x00)
               short ns,           // количество состояний
автомата NODE n, ...);        // список состояний (граф
переходов)};
     bool execute(       // выполнить распознавание цепочки
          FST& fst  // недетерминированный конечный автомат
     );
     FST* automat();}
```

Листинг 3.1 Структура конечного автомата

Структуры пространства имён FST описывают модель недетерминированного конечного автомата, используемого для распознавания цепочек. Узлы графа переходов содержат набор рёбер, каждое из которых определяет переход по символу к следующему состоянию, а функция execute выполняет пошаговую проверку соответствия входной строки заданному автомату.

```
#define GRAPH_NUMBER 8,\
     FST::NODE(1, FST::RELATION('i',1)),\
     FST::NODE(1, FST::RELATION('n',2)),\
     FST::NODE(1, FST::RELATION('t',3)),\
     FST::NODE(1, FST::RELATION('e',4)),\
     FST::NODE(1, FST::RELATION('g',5)),\
     FST::NODE(1, FST::RELATION('e',6)),\
     FST::NODE(1, FST::RELATION('r',7)),\
     FST::NODE()

#define GRAPH_UNSIGNED 9,\
     FST::NODE(1, FST::RELATION('u',1)),\
     FST::NODE(1, FST::RELATION('n',2)),\
     FST::NODE(1, FST::RELATION('s',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('g',5)),\
     FST::NODE(1, FST::RELATION('n',6)),\
     FST::NODE(1, FST::RELATION('e',7)),\
     FST::NODE(1, FST::RELATION('d',8)),\
       FST::NODE()
```

Листинг 3.2 Пример реализации графа конечного автомата для токенов unsigned и integer

Эти макросы определяют конечные автоматы для распознавания ключевых слов языка, таких как integer и unsigned. Каждый автомат представлен последовательностью узлов с переходами по символам, завершающейся пустым узлом для обозначения конца слова.

3.5 Основные структуры данных

Основными структурами данных лексического анализатора являются таблица лексем и таблица идентификаторов. Таблица лексем содержит номер лексемы, лексему (lexema), полученную при разборе, номер строки в исходном коде (sn), и номер в таблице идентификаторов, если лексема является идентификаторов (idxTI). Таблица идентификаторов содержит имя идентификатора (id), номер в таблице лексем (idxfirstLE), тип данных (iddatatype), тип идентификатора (idtype) и значение (или параметры функций) (value). Код C++ со структурой таблицы лексем представлен на листинге 3.3. Код C++ со структурой таблицы идентификаторов представлен на листинге 3.4.

```
struct LEX
{
	LT::LexTable lextable;
	IT::IdTable  idtable;
	LEX() {}
};
struct Graph
{
	char lexema;
	FST::FST graph;
};
```

Листинг 3.3 – Код структуры таблицы лексем

Структура Entry предназначена для хранения информации о лексемах, включая символ, номер состояния, индекс в таблице идентификаторов и при необходимости ссылку на родительский элемент. Структура LexTable представляет собой таблицу лексем с указанием её размера и массивом элементов типа Entry.

```
enum IDDATATYPE { UINT = 1, STR = 2 };
	enum IDTYPE { V = 1, F = 2, P = 3, L = 4, S = 5, U = 6, O =
7 };
	enum SI { Ten = 1, Hex = 2 };

	struct Entry
	{
		int idxfirstLE;
		char id[ID_MAXSIZE];
		IDDATATYPE iddatatype;
		IDTYPE idtype;
		SI si;
		int pars = -1;
		IDDATATYPE* parmstype;
		union
		{
			struct
			{
				unsigned int Ten;
				char Hex[TI_STR_MAXSIZE];
			} vint;
			struct
			{
				char len;
				char str[TI_STR_MAXSIZE];} vstr;
		} value;};

	struct IdTable
	{
		int size;
		Entry* table;};
```

Листинг 3.4 – Код структуры таблицы идентификаторов

Перечисления IDDATATYPE, IDTYPE и SI задают тип данных, тип идентификатора и систему счисления для элементов таблицы идентификаторов. Структуры Entry и IdTable используются для хранения информации об идентификаторах, включая их имя, тип, параметры и значения, а также предоставляют возможность работы с числами и строками через объединение union.

3.6 Структура и перечень сообщений лексического анализатора

Для обработки ошибок лексический анализатор использует таблицу с сообщениями. Структура сообщений содержит информацию о номере сообщения, номер строки и позицию, где было вызвано сообщение в исходном коде, информацию об ошибке. При возникновении сообщения, лексический анализатор игнорирует найденную ошибку и продолжает работу с исходным кодом. Перечень сообщений представлен на рисунке 3.5.

```
ERROR_ENTRY(200, "Lexical error: cannot read from input file (-
in)"),
ERROR_ENTRY(201, "Lexical error: invalid character encoding"),
ERROR_ENTRY(202, "Lexical error: unexpected end of file"),
ERROR_ENTRY(203, "Lexical error: unexpected end of line"),
ERROR_ENTRY(204, "Lexical error: invalid character in string
literal"),
```

Листинг 3.5 – Перечень ошибок лексического анализатора

Листинг 3.5 содержит перечень ошибок лексического анализатора языка RIA-2025, каждая из которых имеет уникальный код и описание. Эти ошибки включают недопустимые символы, неизвестные последовательности, превышение размеров таблиц лексем и идентификаторов, а также превышение длины идентификаторов.

3.7 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Если в процессе анализа происходит ошибка, то анализ останавливается.

3.8 Параметры лексического анализатора

Лексический анализатор принимает обработанный и разбитый на отдельные компоненты исходный код на языке RIA-2025. На выходе формируется таблица лексем и таблица идентификаторов.

Результаты работы лексического анализатора, а именно таблицы лексем и идентификаторов выводятся в файл журнала.

3.9 Алгоритм лексического анализа

– Проверяет входной поток символов программы на исходном языке на допустимость, удаляет лишние пробелы и добавляет сепаратор для вычисления номера строки для каждой лексемы;

– Для выделенной части входного потока выполняется функция распознавания лексемы;

– При успешном распознавании информация о выделенной лексеме заносится в таблицу лексем и таблицу идентификаторов, и алгоритм возвращается к первому этапу;

– Формирует протокол работы;

– При неуспешном распознавании выдается сообщение об ошибке.

Распознавание цепочек основывается на работе конечных автоматов. Работу конечного автомата можно проиллюстрировать с помощью графа переходов. Пример графа для цепочки «intenger» представлен на рисунке 3.3, где S0 – начальное, а S5 – конечное состояние автомата.

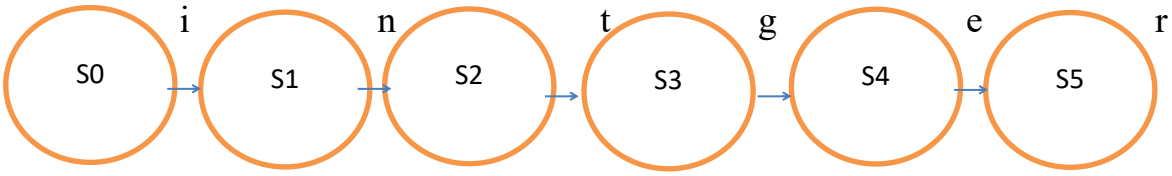


Рисунок 3.3 Пример графа переходов для цепочки string

Определение цепочек выполняется с использованием конечных автоматов, которые проверяют последовательность символов на соответствие шаблону. Граф переходов, показывающий работу такого автомата для слова «integer», представлен на рисунке 3.3, где S0 обозначает начальное, а S5 — конечное состояние.

3.10 Контрольный пример

Результат работы лексического анализатора в виде таблиц лексем и идентификаторов, соответствующих контрольному примеру, представлен в приложении Б.

4 Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализатор: часть компилятора, выполняющая синтаксический анализ, то есть исходный код проверяется на соответствие правилам грамматики. Входной информацией для синтаксического анализа является таблица лексем и таблица идентификаторов. Выходной информацией– дерево разбора

Описание структуры синтаксического анализатора языка представлено на рисунке 4.1.

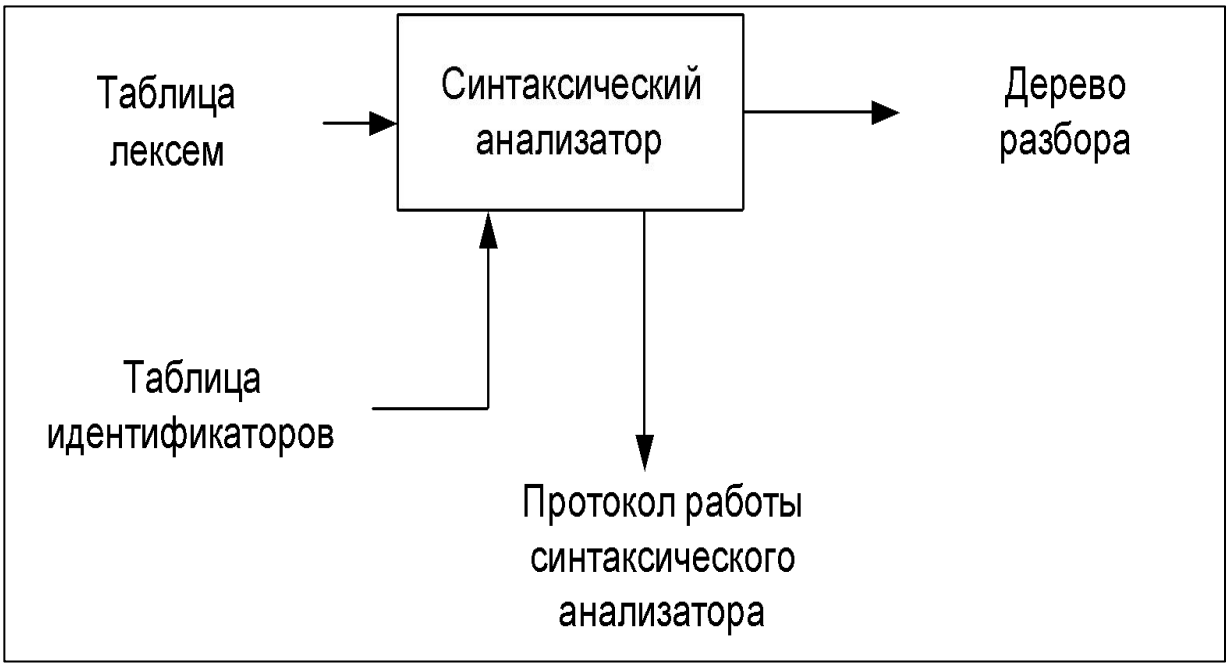


Рисунок 4.1 Структура синтаксического анализатора.

Синтаксический анализатор является компонентом компилятора, который проверяет исходный код на соответствие грамматике языка. В качестве входных данных он использует таблицу лексем и таблицу идентификаторов, а результатом его работы становится дерево разбора, структура которого показана на рисунке 4.1.

4.2 Контекстно свободная грамматика, описывающая синтаксис языка

В синтаксическом анализаторе транслятора языка1 RIA-2025 используется контекстно-свободная грамматика $G = \langle T, N, P, S \rangle$, где

T – множество терминальных символов (было описано в разделе 1.2 данной пояснительной записки),

N – множество нетерминальных символов (первый столбец таблицы 4.1),

P – множество правил языка (второй столбец таблицы 4.1),

S – начальный символ грамматики, являющийся нетерминалом.

Эта грамматика имеет нормальную форму Грейбах, т.к. она не леворекурсивная (не содержит леворекурсивных правил) и правила P имеют вид:

1) $A \to a\alpha$, где $a \quad T, \alpha \quad (T \quad N) \quad \{\lambda\}$; (или $\alpha \quad (T \quad N)^*$, или $\alpha \quad V^*$);

2) $S \to \lambda$, где $S \quad N$ — начальный символ, при этом если такое правило существует, то нетерминал S не встречается в правой части правил.

Описание нетерминальных символов содержится в таблице 4.1.

Таблица 4.1 Таблица правил переходов нетерминальных символов

Символ	Правила	Какие правила порождает
S	S->fti(F){NrE}S S->m{N}	Стартовые правила, описывающее общую структуру программы
N	N->dti;N N->dti; N->i=E;N N->i=E; N-> u(E){N}N N->u(E){N} N->dfti(F);N N->dfti(F); N->pi;N N->pi; N->pl;N N->pl; N->pi(E);N N->pi(E);	Правила для операторов
M	M->vE M->vEM M->sE	Правила для подвыражений
F	F->ti,F F->ti	Правила для списка параметров функции
E	E->i E->l E->iM E->(E) E->i(W) E->i(W)M E->lM	Правила для выражений
W	W->i,W W->l,W W->i W->l W->iM,W W->lM,W W->iM W->lM	Правила для параметров вызываемых функций

Приведённые правила описывают грамматику языка RIA-2025, включая стартовые правила программы, правила для операторов, подвыражений, параметров функций и выражений. Каждая категория правил определяет допустимые конструкции и порядок их использования, обеспечивая корректность синтаксического анализа исходного кода.

4.3 Построение конечного магазинного автомата

Конечный автомат с магазинной памятью представляет собой семерку $M = \langle Q, V, Z, \delta, q_0, z_0, F \rangle$, описание которой представлено в таблице 4.2. Структура данного автомата показана в приложении В

Таблица 4.2 – Описание компонентов магазинного автомата

Компонента	Определение	Описание
Q	Множество состояний автомата	Состояние автомата представляет из себя структуру, содержащую позицию на входной ленте, номера текущего правила и цепочки и стек автомата
V	Алфавит входных символов	Алфавит представляет из себя множества терминальных и нетерминальных символов, описание которых содержится в таблица 3.1 и 4.1.
Z	Алфавит специальных магазинных символов	Алфавит магазинных символов содержит стартовый символ и маркер дна стека (представляет из себя символ $)
δ	Функция переходов автомата	Функция представляет из себя множество правил грамматики, описанных в таблице 4.1.
q_0	Начальное состояние автомата	Состояние, которое приобретает автомат в начале своей работы. Представляется в виде стартового правила грамматики
z_0	Начальное состояние магазина автомата	Символ маркера дна стека $
F	Множество конечных состояний	Конечные состояние заставляют автомат прекратить свою работу. Конечным состоянием является пустой магазин автомата и совпадение позиции на входной ленте автомата с размером ленты

Компоненты автомата включают множество состояний, алфавит входных и специальных магазинных символов, функцию переходов, начальные состояния автомата и магазина, а также множество конечных состояний, определяющих завершение работы. Каждая часть играет свою роль: состояния и функция переходов управляют обработкой входной ленты, а конечные состояния и маркер дна стека обеспечивают корректное завершение работы автомата.

4.4 Основные структуры данных

Основные структуры данных синтаксического анализатора включают в себя структуру магазинного автомата, выполняющего разбор исходной ленты, и структуры грамматики Грейбах, описывающей синтаксические правила языка RIA-2025. Данные структуры представлены в приложении В.

4.5 Описание алгоритма синтаксического разбора

Принцип работы автомата следующий:

– В магазин записывается стартовый символ грамматики;

– На основе полученных ранее таблиц формируется входная лента;

– Запускается автомат;

– Выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;

– Если терминалы в стеке и в ленте совпадают, то данный терминал удаляется из ленты и стека. Иначе возвращаемся в предыдущее сохраненное состояние и выбираем другую цепочку не терминала;

– Если в магазине встретился не терминал, переходим к пункту 4;

– Если наш символ достиг дна стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе генерируется исключение. После 3 исключений синтаксический анализатор завершает свою работу.

4.6 Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора представлен на листинге 4.1.

```
ERROR_ENTRY(600, "Syntax error: unexpected end of program"),
ERROR_ENTRY(601, "Syntax error: expected identifier after
keyword"),
ERROR_ENTRY(602, "Syntax error: error in function declaration"),
ERROR_ENTRY(603, "Syntax error: error in variable declaration"),
ERROR_ENTRY(604, "Syntax error: error in expression parsing"),
```

Листинг 4.1 – Перечень сообщений синтаксического анализатора

Перечень ошибок синтаксического анализатора включает коды 600–605, каждый из которых соответствует определённой проблеме в структуре программы или функции. Эти ошибки помогают выявлять неправильный синтаксис, ошибки в списках параметров, теле функций и при вызове функций или выражений.

4.7 Параметры синтаксического анализатора и режимы его работы

Входной информацией для синтаксического анализатора является таблица лексем и идентификаторов. Результаты работы лексического разбора,

а именно дерево разбора и протокол работы автомата с магазинной памятью выводятся в журнал работы программы.

4.8 Принцип обработки ошибок

Обработка ошибок происходит следующим образом:

– Синтаксический анализатор перебирает все правила и цепочки правила грамматики для нахождения подходящего соответствия с конструкцией, представленной в таблице лексем.

– Если невозможно подобрать подходящую цепочку, то генерируется соответствующая ошибка.

– В случае нахождения ошибки, после всей процедуры трассировки в протокол будет выведено диагностическое сообщение.

В структуре грамматики Грейбах цепочки в правилах расположены в порядке приоритета, самые часто используемые располагаются выше, а те, что используются реже – ниже.

4.9 Контрольный пример

Пример разбора синтаксическим анализатором исходного кода на языке RIA-2025 представлен в приложении В. Дерево разбора исходного кода также представлено в приложении В.

5 Разработка семантического анализатора

5.1 Структура семантического анализатора

Семантический анализатор принимает на свой вход результаты работ лексического и синтаксического анализаторов, то есть таблицы лексем, идентификаторов и результат работы синтаксического анализатора, то есть дерево разбора, и последовательно ищет необходимые ошибки. Некоторые проверки (такие как проверка на единственность точки входа, проверка на предварительное объявление переменной) осуществляются в процессе лексического анализа. Общая структура обособленно работающего (не параллельно с лексическим анализом) семантического анализатора представлена на рисунке 5.1.

Рисунок 5.1. Структура семантического анализатора

Семантический анализатор использует результаты лексического и синтаксического анализов, включая таблицы лексем, идентификаторов и дерево разбора, для последовательного поиска семантических ошибок. Некоторые проверки, например на уникальность точки входа или предварительное объявление переменной, могут выполняться ещё на этапе лексического анализа; общая структура автономного семантического анализатора показана на рисунке 5.1.

5.2 Функции семантического анализатора

Семантический анализатор выполняет проверку на основные правила языка (семантики языка), которые описаны в разделе 1.16. Информация об ошибках выводится в консоль, а также в протокол работы.

5.3 Структура и перечень сообщений семантического анализатора

Семантический анализатор проверяет корректность смысловой структуры программы, используя информацию из лексического и синтаксического анализов. Он анализирует таблицы лексем, идентификаторов и дерево разбора для выявления ошибок логики и

соответствия правил языка. Сообщения, формируемые семантическим анализатором, представлены на листинге 5.1.

```
    ERROR_ENTRY(300, "Semantic error: undefined identifier"),
ERROR_ENTRY(301, "Semantic error: missing main function
declaration"),
ERROR_ENTRY(302, "Semantic error: duplicate main function
declaration"),
ERROR_ENTRY(303, "Semantic error: function has no return
statement"),
ERROR_ENTRY(304, "Semantic error: function has incorrect return
type"),
ERROR_ENTRY(305, "Semantic error: missing function parameter
declaration"),
```

Листинг 5.1 – Перечень сообщений семантического анализатора

После обнаружения ошибок семантический анализатор формирует отчёт, который используется для последующей отладки и корректировки исходного кода. Такой подход обеспечивает надёжность работы компилятора и предотвращает выполнение некорректных программ.

5.4 Принцип обработки ошибок

Ошибки, возникающие в процессе трансляции программы, фиксируются в протокол, заданный входным параметрами. В случае возникновения ошибок происходит их протоколирование с номером ошибки и диагностическим сообщением. Анализ останавливается после того, как будут найдены все ошибки.

5.5 Контрольный пример

Соответствие примеров некоторых ошибок в исходном коде и диагностических сообщений об ошибках приведено в таблице 5.1.

Таблица 5.1. Примеры диагностики ошибок

Исходный код	Текст сообщения

main
{
declare number x;
x = 9;
new string y;
y=x;
}
	Ошибка N309: Семантическая ошибка:
	Типы данных в выражении не совпадают
	Строка: 6

main{ new number x; x = 9;} main { declare string y; y = "qwerty";}	Ошибка N302: Семантическая ошибка: Обнаружено несколько точек входа main Строка: 4

Примеры исходного кода на языке RIA-2025 наглядно иллюстрируют различные ситуации, при которых семантический анализатор выявляет

ошибки в структуре и логике программы. Эти примеры позволяют продемонстрировать типичные нарушения правил языка, возникающие на этапе смыслового анализа кода. В частности, подробно рассматриваются случаи несовпадения типов данных в выражениях, когда выполняются недопустимые операции над несовместимыми типами, например попытка сложения числового и строкового значений. Подобные ошибки делают дальнейшее выполнение программы невозможным и должны быть обнаружены и устранены ещё на этапе трансляции.

Кроме того, в представленных примерах демонстрируются ситуации обнаружения нескольких точек входа main, что является серьёзным нарушением семантических правил языка RIA-2025. Наличие более одной главной функции приводит к неоднозначности начала выполнения программы и поэтому запрещено. При возникновении подобных ошибок транслятор формирует подробные диагностические сообщения с указанием номера строки и характера ошибки в исходном коде. Это позволяет пользователю быстро определить проблемный участок программы, понять причину ошибки и внести необходимые исправления.

6 Преобразование выражений

6.1 Выражения, допускаемые языком

В языке RIA-2025 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. Приоритеты операций

Операция	Значение приоритета
( )	3
*	2
/	2
+	1
-	1

Каждая арифметическая операция имеет свой приоритет, который определяет порядок её выполнения в выражении. Скобки ( ) имеют наивысший приоритет, затем идут умножение * и деление /, а сложение + и вычитание - имеют наименьший приоритет.

6.2 Польская запись и принцип ее построения

Выражения в языке RIA-2025 преобразовываются к обратной польской записи.

Польская запись – это альтернативный способ записи арифметических выражений, преимущество которого состоит в отсутствии скобок.

Обратная польская запись – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

Алгоритм построения:

– исходная строка: выражение;

– результирующая строка: польская запись;

– стек: пустой;

– результирующая строка: польская запись;

– исходная строка просматривается слева направо;

– операнды переносятся в результирующую строку в порядке их следования;

– операция записывается в стек, если стек пуст или в вершине стека лежит отрывающая скобка;

– операция выталкивает все операции с большим или равным приоритетом в результирующую строку;

– запятая не помещается в стек, если в стеке операции, то все выбираются в строку;

– отрывающая скобка помещается в стек;

– закрывающая скобка выталкивает все операции до открывающей скобки, после чего обе скобки уничтожаются;

– закрывающая скобка с приоритетом, равным 4, выталкивает все до открывающей с таким же приоритетом и генерирует @ (@ – специальный символ, в который записывается информация о вызываемой функции), а в поле приоритета для данной лексемы записывается число параметров вызываемой функции;

– по итогам разбора исходной строки все операции, оставшиеся в стеке, выталкиваются в результирующую строку.

Использование польской записи позволяет вычислить выражение за один проход.

Таблица 6.2 – Пример преобразования выражения в обратную польскую запись

Исходная строка	Результирующая строка
i=i+i-i+i;	i=iiii+-+;
i=i+l-l;	i=ill+-;
i=i(i,i)+l;	i=ii@2il+;

Примеры показывают преобразование исходных выражений в результирующую строку в процессе синтаксического анализа или подготовки к польской записи. При этом идентификаторы и литералы заменяются на внутренние обозначения, а операции сохраняют порядок действий с соответствующими символами.

6.3 Программная реализация обработки выражений

Программная реализация алгоритма преобразования выражений к польской записи представлена в приложении Г.

6.4 Контрольный пример

В приложении Г приведен результат преобразования выражений в польский формат.

7 Генерация кода

7.1 Структура генератора кода

В языке RIA-2025 генерация кода является заключительным этапом трансляции. Генератор принимает на вход таблицы лексем и идентификаторов, полученные в результате лексического анализа. В соответствии с таблицей лексем строится выходной файл на языке ассемблера, который будет являться результатом работы транслятора. В случае возникновения ошибок генерация кода не будет осуществляться. Структура генератора кода RIA-2025 представлена на рисунке 7.1.

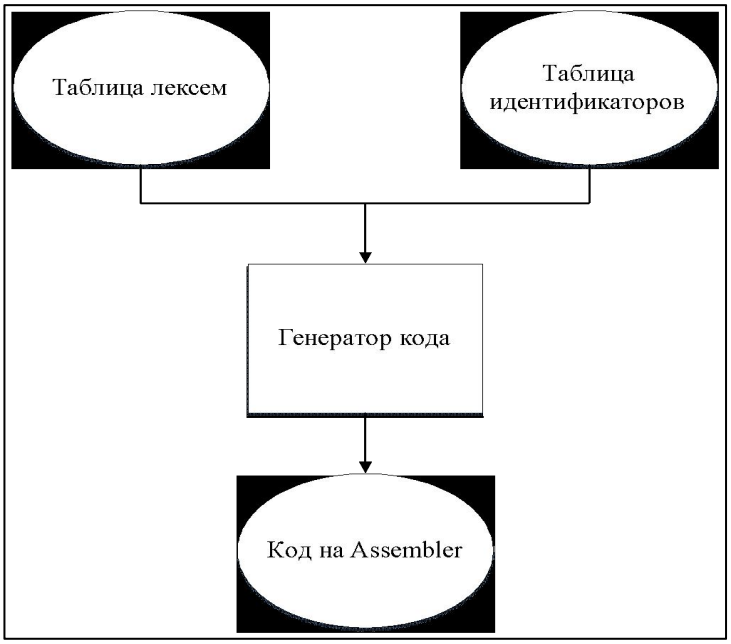


Рисунок 7.1 – Структура генератора кода

Этап генерации кода в RIA-2025 отвечает за преобразование промежуточного представления программы в ассемблерный код. На этом этапе генератор использует результаты лексического и синтаксического анализа, и в случае обнаружения ошибок создание кода не выполняется.

7.2 Представление типов данных в оперативной памяти

Элементы таблицы идентификаторов расположены в сегментах .data и .const языка ассемблера. Соответствия между типами данных идентификаторов на языке RIA-2025 и на языке ассемблера приведены в таблице 7.1. Каждому идентификатору присваивается уникальный адрес в памяти, что обеспечивает корректную работу программы после трансляции. Значения констант помещаются в сегмент .const, а изменяемые переменные — в сегмент .data. Такой подход позволяет генератору кода создавать корректные инструкции и обращаться к данным по их адресам.

Таблица 7.1 – Соответствия типов идентификаторов языка RIA-2025 и языка ассемблера

Тип идентификатора на языке RIA-2025	Тип идентификатора на языке ассемблера	Пояснение
integer	dword	Хранит целочисленный тип данных
unsigned	dword	Хранит беззнаковый целый тип данных
string	dword	Хранит указатель на начало строки. Строка завешается нулевым символом
bool	dword	Тип данных, хранящий логическое значение

В языке ассемблера все типы идентификаторов языка RIA-2025 представлены как dword. При этом integer и unsigned хранят целые и беззнаковые числа соответственно, string — указатель на начало строки с нулевым завершением, а bool — логическое значение.

7.3 Статическая библиотека

В языке RIA-2025 предусмотрена статическая библиотека. Статическая библиотека содержит функции, написанные на языке C++. Объявление функций статической библиотеки генерируется автоматически в коде ассемблера. Объявление функций статической библиотеки генерируется автоматически.

Таблица 7.2 – Функции статической библиотеки

Функция	Назначение
void prints(char* str)	Вывод на консоль строки str
void printu(int ui)	Вывод на консоль целочисленной беззнаковой переменной ui
unsigned int sravs(char* str1, char* str2)	Сравнение строк
unsigned int stepen(unsigned int ui1, unsigned int ui2)	Возведение числа ui1 в степень ui2

Функции prints и printu предназначены для вывода на консоль строк и беззнаковых целых чисел соответственно. Функции sravs и stepen выполняют сравнение строк и возведение числа в степень, возвращая результат в виде беззнакового целого значения.

7.4 Особенности алгоритма генерации кода

В языке RIA-2025 генерация кода строится на основе таблиц лексем и идентификаторов. Общая схема работы генератора кода представлена на рисунке 7.2.

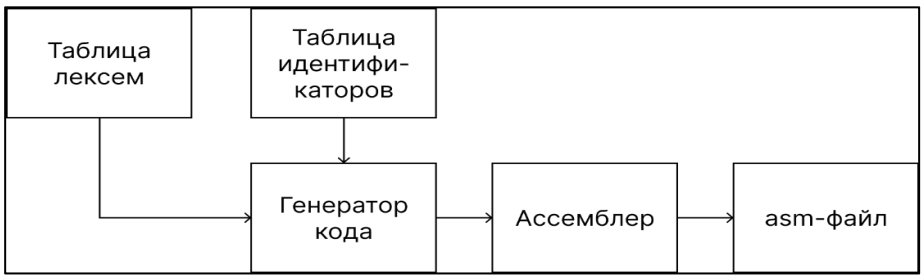


Рисунок 7.2 – Общая схема работы генератора кода

В RIA-2025 генератор кода использует таблицы лексем и идентификаторов для построения ассемблерного кода. Структура и порядок работы генератора показаны на рисунке 7.2.

7.5 Входные параметры генератора кода

На вход генератору кода поступают таблицы лексем и идентификаторов исходного код программы на языке RIA-2025. Результаты работы генератора кода выводятся в файл с расширением .asm.

7.6 Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении Д. Результат работы контрольного примера также приведён в приложении Д.

8 Тестирование транслятора

8.1 Общие положения

В языке RIA-2025 не разрешается использовать запрещённые входным алфавитом символы. Результат использования запрещённого символа показан в таблице 8.1.

Таблица 8.1 – Тестирование фазы проверки на допустимость символов

Исходный код	Диагностическое сообщение

main
{
@
}
	Ошибка 111: Недопустимый символ в исходном файле (-in), строка 3 ,позиция 2.

Пример исходного кода демонстрирует ситуацию, когда в программе встречается недопустимый символ @. Диагностическое сообщение указывает на ошибку с кодом 111, сообщая строку и позицию символа в исходном файле.

8.2 Результаты тестирования

На этапе лексического анализа могут возникнуть ошибки, описанные в пункте 3.7.

Результаты тестирования анализатора показаны в таблице 8.2.

Таблица 8.2 – Тестирование лексического анализатора

Исходный код	Диагностическое сообщение
main {declare string x11;}	Error 201: Lexical error: invalid character encoding, line 13, column 0
function string fi(string id, string ig) {return id;} main {declare string str; str=fi(i;i);}	Error 200: Lexical error: cannot read from input file (-in), line 26, column 33
function unsigned integer lililililililililili(unsigned a) {…}	Error 204: Lexical error: invalid character in string literal, line 14, column 0

На этапе лексического анализа обрабатываются ошибки, которые препятствую правильному построению таблицы лексем и идентификаторов.

На этапе синтаксического анализа в языке RIA-2025 могут возникнуть ошибки, описанные в пункте 4.6. Результаты тестирования синтаксического анализатора показаны в таблице 8.3.

Таблица 8.3 - Тестирование синтаксического анализатора

Исходный код	Диагностическое сообщение
1	2
main new number x; }	Error 610: line 9, Syntax error: missing opening parenthesis

Продолжение таблицы 8.3

1	2
function string fi(string id) {declare string ig; ig= +id}	Error 614: line 79, Syntax error: invalid operator precedence
main { declare unsigned num; num=1; declare unsigned nums; nums=num++num;}	Error 614: line 27, Syntax error: invalid operator precedence

Семантический анализ в языке RIA-2025 содержит множество проверок по семантическим правилам, описанным в пункте 1.16. Итоги тестирования семантического анализатора на корректное обнаружение семантических ошибок приведены в таблице 8.4.

Таблица 8.4 -Тестирование семантического анализатора

Исходный код	Диагностическое сообщение
function string fi(string str){return 5;} main{return 0;}	Error 315: Semantic error: function call in expression context, line 2, column 0
Main{declare unsigned x; x = 5 + "abc";	Error 317: Semantic error: invalid type conversion, line 78, column 0
function string fi(string par){return par;} main{declare string str; str=fi(“a”,”b”,”c”,”d”);}	Error 308: Semantic error: function call with incorrect number of arguments, line 26, column 0
main{declare string x; x = "abc" + "d";}	Error 316: Semantic error: assignment to constant variable, line 20, column 0
main {declare unsigned integer x; x=5; if(lex==”gf”){cout 5;}}	Error 317: Semantic error: invalid type conversion, line 45, column 0
main {declare unsigned integer x; x=5/0}	Error 318: Semantic error: division by zero, line 78, column 0
main { a = 7; }	Error 300: Semantic error: undefined identifier, line 22, column 0
declare integer s; s = 5 + 4;	Error 301: Semantic error: missing main function declaration, line 0, column 0
main { } main {}	Error 302: Semantic error: duplicate main function declaration, line 0, column 0

Примеры исходного кода показывают различные семантические ошибки, которые может выявить анализатор языка RIA-2025. Диагностические сообщения указывают код ошибки, строку и характер нарушения, что облегчает поиск и исправление ошибок в программе.

Заключение

В ходе выполнения курсовой работы был разработан транслятор для языка программирования RIA-2025. Таким образом, были выполнены основные задачи данной курсовой работы:

- Сформулирована спецификация языка RIA-2025;
- Разработаны конечные автоматы и алгоритмы для реализация лексического анализатора;
- Разработана контекстно-свободная, приведённая к нормальной форме Грейбах, грамматика для описания синтаксически верных конструкций языка;
- Разработан семантический анализатор, осуществляющий проверку смысла используемых инструкций;
- Разработан транслятор с языка программирования RIA-2025 на язык низкого уровня Assembler;
- Проведено тестирование всех вышеперечисленных компонентов.

Окончательная версия языка RIA-2025 включает:

– 4 типа данных;
– Поддержка операции вывода;
– 2 библиотечные функции
– Возможность вызова функций стандартной библиотеки;
– Наличие 4 арифметических операторов для вычисления выражений;
– Наличие 6 операторов сравнения для целочисленных переменных
– Структурированная система для обработки ошибок пользователя.
– Условный оператор;

Язык RIA-2025 предоставляет минимальный, но достаточный набор возможностей, включающий два типа данных, базовые арифметические и логические операции, а также поддержку вывода информации. Он позволяет вызывать стандартные библиотечные функции и использовать условный оператор для организации ветвления. Кроме того, язык оснащён структурированной системой обработки ошибок, что улучшает надёжность разработки и упрощает диагностику.

Список использованных источников

1. Курс лекций по предмету «Конструирование программного обеспечения» Наркевич А.С.

2. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2003. – 768с.

3. Молчанов, А. Ю. Системное программное обеспечение / А. Ю. Молчанов. – СПб.: Питер, 2010. – 400 с.

4. Ахо, А. Теория синтаксического анализа, перевода и компиляции /А. Ахо, Дж. Ульман. – Москва : Мир, 1998. – Т. 2 : Компиляция. - 487 с.

5. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

6. Орлов, С.А. Теория и практика языков программирования / С.А. Орлов – 2014. – 689 с.

7. Страуструп, Б. Принципы и практика использования C++ / Б. Страуструп – 2009 – 1238 с.

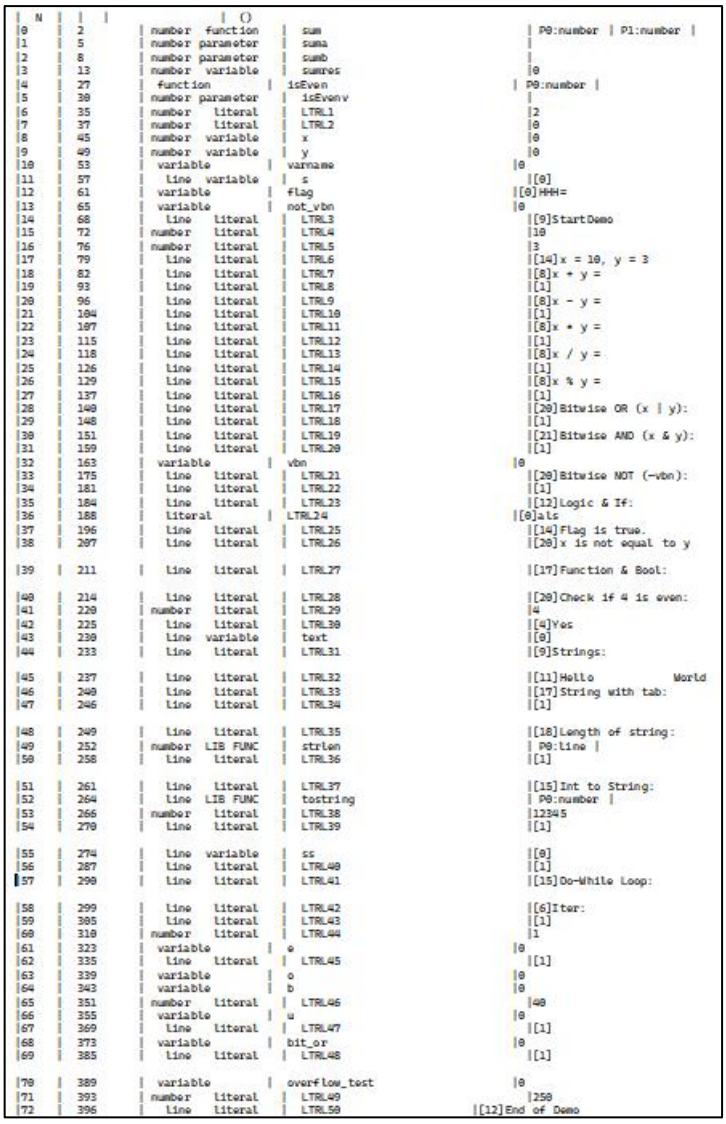
Приложение А

```
Function integer sum(integer a, integer b) {
    declare integer res;
    res = a + b;
    return res;}
function bool isEven(integer v) {
    return (v % 2) == 0;}
main {
    declare integer x;
    declare integer y;
    declare unsigned varname;
    declare string s;
    declare bool flag;
    declare unsigned not_vbn;
    cout "StartDemo";
    x = 10;
    y = 3;
    cout "x = 10, y = 3\n";
    cout "x + y = "; cout sum(x, y); cout "\n";
    cout "x - y = "; cout x - y; cout "\n";
    cout "x * y = "; cout x * y; cout "\n";
    cout "x / y = "; cout x / y; cout "\n";
    cout "x % y = "; cout x % y; cout "\n";
    cout "Bitwise OR (x | y): "; cout x | y; cout "\n";
    cout "Bitwise AND (x & y): "; cout x & y; cout "\n";
    declare unsigned vbn;
    vbn = 3;
    not_vbn = ~vbn;
    cout "Bitwise NOT (~vbn): "; cout not_vbn; cout "\n";
    cout "Logic & If:\n";
    flag = false;
    if (!(flag)) {
        cout "Flag is true.\n";}
    if (x != y) {
        cout "x is not equal to y\n"; }
    cout "Function & Bool:\n";
    cout "Check if 4 is even: ";
    if (isEven(4)) {
        cout "Yes\n";}
    declare string text;
    cout "Strings:\n";
    s = "Hello\tWorld";
    cout "String with tab: "; cout s; cout "\n";
    cout "Length of string: "; cout strlen(s); cout "\n";
    cout "Int to String: "; cout tostring(12345); cout "\n";
declare string ss;
ss = tostring(x);
cout ss; cout "\n";
    cout "Do-While Loop:\n";
    x = 0;
    do {cout "Iter: "; cout x; cout "\n";
```

```
        x = x + 1;} while (x != 3);
declare unsigned e;
e = 0;
cout 2 + e; cout "\n";
declare unsigned o;
declare unsigned b;
o = 10;
b = 40;
declare unsigned u;
u = (o + b / 2);
cout u; cout "\n";
declare unsigned bit_or;
bit_or = o | b;
cout bit_or; cout "\n";
declare unsigned overflow_test;
overflow_test = 250;
cout "End of Demo\n";}
```

Листинг 1 - Исходный код на языке RIA-2025

Приложение Б



N			()		
0	2	number function	sum	P0:number P1:number	
1	5	number parameter	suma		
2	8	number parameter	sumb		
3	13	number variable	sumres	0	
4	27	function	isEven	P0:number	
5	30	number parameter	isEvenv		
6	35	number literal	LTRL1	2	
7	37	number literal	LTRL2	0	
8	45	number variable	x	0	
9	49	number variable	y	0	
10	53	variable	varname	0	
11	57	line variable	s	[0]	
12	61	variable	flag	[0]HHH=	
13	65	variable	not_vbn	0	
14	68	line literal	LTRL3	[9]StartDemo	
15	72	number literal	LTRL4	10	
16	76	number literal	LTRL5	3	
17	79	line literal	LTRL6	[14]x = 10, y = 3	
18	82	line literal	LTRL7	[8]x + y =	
19	93	line literal	LTRL8	[1]	
20	96	line literal	LTRL9	[8]x - y =	
21	104	line literal	LTRL10	[1]	
22	107	line literal	LTRL11	[8]x * y =	
23	115	line literal	LTRL12	[1]	
24	118	line literal	LTRL13	[8]x / y =	
25	126	line literal	LTRL14	[1]	
26	129	line literal	LTRL15	[8]x % y =	
27	137	line literal	LTRL16	[1]	
28	140	line literal	LTRL17	[20]Bitwise OR (x | y):	
29	148	line literal	LTRL18	[1]	
30	151	line literal	LTRL19	[21]Bitwise AND (x & y):	
31	159	line literal	LTRL20	[1]	
32	163	variable	vbn	0	
33	175	line literal	LTRL21	[20]Bitwise NOT (~vbn):	
34	181	line literal	LTRL22	[1]	
35	184	line literal	LTRL23	[12]Logic & If:	
36	188	literal	LTRL24	[0]als	
37	196	line literal	LTRL25	[14]Flag is true.	
38	207	line literal	LTRL26	[20]x is not equal to y	
39	211	line literal	LTRL27	[17]Function & Bool:	
40	214	line literal	LTRL28	[20]Check if 4 is even:	
41	220	number literal	LTRL29	4	
42	225	line literal	LTRL30	[4]Yes	
43	230	line variable	text	[0]	
44	233	line literal	LTRL31	[9]Strings:	
45	237	line literal	LTRL32	[11]Hello World	
46	248	line literal	LTRL33	[17]String with tab:	
47	246	line literal	LTRL34	[1]	
48	249	line literal	LTRL35	[18]Length of string:	
49	252	number LIB FUNC	strlen	P0:line	
50	258	line literal	LTRL36	[1]	
51	261	line literal	LTRL37	[15]Int to String:	
52	264	line LIB FUNC	tostring	P0:number	
53	266	number literal	LTRL38	12345	
54	270	line literal	LTRL39	[1]	
55	274	line variable	ss	[0]	
56	287	line literal	LTRL40	[1]	
57	290	line literal	LTRL41	[15]Do-While Loop:	
58	299	line literal	LTRL42	[6]Iter:	
59	305	line literal	LTRL43	[1]	
60	310	number literal	LTRL44	1	
61	323	variable	e	0	
62	335	line literal	LTRL45	[1]	
63	339	variable	o	0	
64	343	variable	b	0	
65	351	number literal	LTRL46	48	
66	355	variable	u	0	
67	369	line literal	LTRL47	[1]	
68	373	variable	bit_or	0	
69	385	line literal	LTRL48	[1]	
70	389	variable	overflow_test	0	
71	393	number literal	LTRL49	250	
72	396	line literal	LTRL50	[12]End of Demo	

Рисунок 1 - Таблица идентификаторов на выходе лексического анализатора

N	ЛЕКСЕМА	СТРОКА	ИНДЕКС В ТИ
0	f	1	
1	t	1	
2	i	1	0
3	(	1	
4	t	1	
5	i	1	1
6	,	1	
7	t	1	
8	i	1	2
9	)	1	
10	{	1	
11	v	2	
12	t	2	
13	i	2	3
14	;	2	
15	i	3	3
16	=	3	
17	i	3	1
18	i	3	2
19	+	3	
20	;	3	
21	e	4	
22	i	4	3
23	;	4	
24	}	5	
25	f	7	
26	t	7	
27	i	7	4
28	(	7	
29	t	7	
30	i	7	5
31	)	7	
32	{	7	
33	e	8	

Рисунок 2 – Часть таблицы лексем

Приложение В

```
#pragma once
#include "FST.h"
#define N_GRAPHS 21

// Separators graph
#define GRAPH_SEPARATORS 3,\
     FST::NODE(28,\
               FST::RELATION(';',2), FST::RELATION('=',2),
FST::RELATION('=',1),\
               FST::RELATION(',',2), FST::RELATION('[',2),\
               FST::RELATION(']',2), FST::RELATION('(',2),\
               FST::RELATION(')',2), FST::RELATION('*',2),\
               FST::RELATION('+',2), FST::RELATION('-',2),\
               FST::RELATION('/',2), FST::RELATION('%',2),\
               FST::RELATION('{',2), FST::RELATION('}',2),\
               FST::RELATION('<',2), FST::RELATION('<',1),\
               FST::RELATION('>',2), FST::RELATION('>',1),\
               FST::RELATION('|',2), FST::RELATION('&',2),\
               FST::RELATION('~',2), FST::RELATION('!',2),
FST::RELATION('!',1),\
               FST::RELATION('?',2), FST::RELATION('^',2),\
            FST::RELATION('@',2), FST::RELATION('#',2)),\
     FST::NODE(1, FST::RELATION('=', 2)), \
     FST::NODE()


#define GRAPH_ID 2, \
     FST::NODE(53,FST::RELATION('a',1),  FST::RELATION('b',1), \
               FST::RELATION('c', 1),  FST::RELATION('d', 1), \
               FST::RELATION('e', 1),  FST::RELATION('f', 1), \
               FST::RELATION('g', 1),  FST::RELATION('h', 1), \
               FST::RELATION('i', 1),  FST::RELATION('j', 1), \
               FST::RELATION('k', 1),  FST::RELATION('l', 1), \
               FST::RELATION('m', 1),  FST::RELATION('n', 1), \
               FST::RELATION('o', 1),  FST::RELATION('p', 1), \
               FST::RELATION('q', 1),  FST::RELATION('r', 1), \
               FST::RELATION('s', 1),  FST::RELATION('t', 1), \
               FST::RELATION('u', 1),  FST::RELATION('v', 1), \
               FST::RELATION('w', 1),  FST::RELATION('x', 1), \
               FST::RELATION('y', 1),  FST::RELATION('z', 1), \
               FST::RELATION('A', 1),  FST::RELATION('B', 1), \
               FST::RELATION('C', 1),  FST::RELATION('D', 1), \
               FST::RELATION('E', 1),  FST::RELATION('F', 1), \
               FST::RELATION('G', 1),  FST::RELATION('H', 1), \
               FST::RELATION('I', 1),  FST::RELATION('J', 1), \
               FST::RELATION('K', 1),  FST::RELATION('L', 1), \
               FST::RELATION('M', 1),  FST::RELATION('N', 1), \
               FST::RELATION('O', 1),  FST::RELATION('P', 1), \
               FST::RELATION('Q', 1),  FST::RELATION('R', 1), \
               FST::RELATION('S', 1),  FST::RELATION('T', 1), \
               FST::RELATION('U', 1),  FST::RELATION('V', 1), \
```

```
                FST::RELATION('W', 1),  FST::RELATION('X', 1), \
                FST::RELATION('Y', 1),  FST::RELATION('Z', 1), \
              FST::RELATION('_', 1)), \
     FST::NODE(53, FST::RELATION('a',1),  FST::RELATION('b',1), \
                FST::RELATION('c', 1),  FST::RELATION('d', 1), \
                FST::RELATION('e', 1),  FST::RELATION('f', 1), \
                FST::RELATION('g', 1),  FST::RELATION('h', 1), \
                FST::RELATION('i', 1),  FST::RELATION('j', 1), \
                FST::RELATION('k', 1),  FST::RELATION('l', 1), \
                FST::RELATION('m', 1),  FST::RELATION('n', 1), \
                FST::RELATION('o', 1),  FST::RELATION('p', 1), \
                FST::RELATION('q', 1),  FST::RELATION('r', 1), \
                FST::RELATION('s', 1),  FST::RELATION('t', 1), \
                FST::RELATION('u', 1),  FST::RELATION('v', 1), \
                FST::RELATION('w', 1),  FST::RELATION('x', 1), \
                FST::RELATION('y', 1),  FST::RELATION('z', 1), \
                FST::RELATION('A', 1),  FST::RELATION('B', 1), \
                FST::RELATION('C', 1),  FST::RELATION('D', 1), \
                FST::RELATION('E', 1),  FST::RELATION('F', 1), \
                FST::RELATION('G', 1),  FST::RELATION('H', 1), \
                FST::RELATION('I', 1),  FST::RELATION('J', 1), \
                FST::RELATION('K', 1),  FST::RELATION('L', 1), \
                FST::RELATION('M', 1),  FST::RELATION('N', 1), \
                FST::RELATION('O', 1),  FST::RELATION('P', 1), \
                FST::RELATION('Q', 1),  FST::RELATION('R', 1), \
                FST::RELATION('S', 1),  FST::RELATION('T', 1), \
                FST::RELATION('U', 1),  FST::RELATION('V', 1), \
                FST::RELATION('W', 1),  FST::RELATION('X', 1), \
                FST::RELATION('Y', 1),  FST::RELATION('Z', 1), \
              FST::RELATION('_', 1))

#define GRAPH_STRING_LITERAL 3,\
     FST::NODE(1, FST::RELATION('\"', 1)),\
     FST::NODE(89, \
                FST::RELATION('a', 1), FST::RELATION('b', 1),
FST::RELATION('c', 1), FST::RELATION('d', 1),\
                FST::RELATION('e', 1), FST::RELATION('f', 1),
FST::RELATION('g', 1), FST::RELATION('h', 1),\
                FST::RELATION('i', 1), FST::RELATION('j', 1),
FST::RELATION('k', 1), FST::RELATION('l', 1),\
                FST::RELATION('m', 1), FST::RELATION('n', 1),
FST::RELATION('o', 1), FST::RELATION('p', 1),\
                FST::RELATION('q', 1), FST::RELATION('r', 1),
FST::RELATION('s', 1), FST::RELATION('t', 1),\
                FST::RELATION('u', 1), FST::RELATION('v', 1),
FST::RELATION('w', 1), FST::RELATION('x', 1),\
                FST::RELATION('y', 1), FST::RELATION('z', 1),\
                FST::RELATION('A', 1), FST::RELATION('B', 1),
FST::RELATION('C', 1), FST::RELATION('D', 1),\
                FST::RELATION('E', 1), FST::RELATION('F', 1),
FST::RELATION('G', 1), FST::RELATION('H', 1),\
                FST::RELATION('I', 1), FST::RELATION('J', 1),
FST::RELATION('K', 1), FST::RELATION('L', 1),\
```

```
                FST::RELATION('M', 1), FST::RELATION('N', 1),
FST::RELATION('O', 1), FST::RELATION('P', 1),\
                FST::RELATION('Q', 1), FST::RELATION('R', 1),
FST::RELATION('S', 1), FST::RELATION('T', 1),\
                FST::RELATION('U', 1), FST::RELATION('V', 1),
FST::RELATION('W', 1), FST::RELATION('X', 1),\
                FST::RELATION('Y', 1), FST::RELATION('Z', 1),\
                FST::RELATION('0', 1), FST::RELATION('1', 1),
FST::RELATION('2', 1), FST::RELATION('3', 1),\
                FST::RELATION('4', 1), FST::RELATION('5', 1),
FST::RELATION('6', 1), FST::RELATION('7', 1),\
                FST::RELATION('8', 1), FST::RELATION('9', 1),\
                FST::RELATION(' ', 1), FST::RELATION(',', 1),
FST::RELATION('.', 1), FST::RELATION(';', 1),\
                FST::RELATION('-', 1), FST::RELATION('+', 1),
FST::RELATION('*', 1), FST::RELATION('/', 1),\
                FST::RELATION('=', 1), FST::RELATION(':', 1),
FST::RELATION(')', 1), FST::RELATION('(', 1),\
                FST::RELATION('}', 1), FST::RELATION('{', 1),
FST::RELATION(']', 1), FST::RELATION('[', 1),\
                FST::RELATION('!', 1), FST::RELATION('?', 1),
FST::RELATION('#', 1), FST::RELATION('&', 1),\
                FST::RELATION('>', 1), FST::RELATION('<', 1),
FST::RELATION('%', 1),\
                FST::RELATION('\\', 1), FST::RELATION('|', 1),
FST::RELATION('~', 1), FST::RELATION('\"', 2)), \
     FST::NODE()

#define GRAPH_CHAR_LITERAL 3,\
     FST::NODE(1, FST::RELATION('\'', 1)),\
     FST::NODE(89, \
                FST::RELATION('a', 1), FST::RELATION('b', 1),
FST::RELATION('c', 1), FST::RELATION('d', 1),\
                FST::RELATION('e', 1), FST::RELATION('f', 1),
FST::RELATION('g', 1), FST::RELATION('h', 1),\
                FST::RELATION('i', 1), FST::RELATION('j', 1),
FST::RELATION('k', 1), FST::RELATION('l', 1),\
                FST::RELATION('m', 1), FST::RELATION('n', 1),
FST::RELATION('o', 1), FST::RELATION('p', 1),\
                FST::RELATION('q', 1), FST::RELATION('r', 1),
FST::RELATION('s', 1), FST::RELATION('t', 1),\
                FST::RELATION('u', 1), FST::RELATION('v', 1),
FST::RELATION('w', 1), FST::RELATION('x', 1),\
                FST::RELATION('y', 1), FST::RELATION('z', 1),\
                FST::RELATION('A', 1), FST::RELATION('B', 1),
FST::RELATION('C', 1), FST::RELATION('D', 1),\
                FST::RELATION('E', 1), FST::RELATION('F', 1),
FST::RELATION('G', 1), FST::RELATION('H', 1),\
                FST::RELATION('I', 1), FST::RELATION('J', 1),
FST::RELATION('K', 1), FST::RELATION('L', 1),\
                FST::RELATION('M', 1), FST::RELATION('N', 1),
FST::RELATION('O', 1), FST::RELATION('P', 1),\
                FST::RELATION('Q', 1), FST::RELATION('R', 1),
```

```
FST::RELATION('S', 1), FST::RELATION('T', 1),\
               FST::RELATION('U', 1), FST::RELATION('V', 1),
FST::RELATION('W', 1), FST::RELATION('X', 1),\
               FST::RELATION('Y', 1), FST::RELATION('Z', 1),\
               FST::RELATION('0', 1), FST::RELATION('1', 1),
FST::RELATION('2', 1), FST::RELATION('3', 1),\
               FST::RELATION('4', 1), FST::RELATION('5', 1),
FST::RELATION('6', 1), FST::RELATION('7', 1),\
               FST::RELATION('8', 1), FST::RELATION('9', 1),\
               FST::RELATION(' ', 1), FST::RELATION(',', 1),
FST::RELATION('.', 1), FST::RELATION(';', 1),\
               FST::RELATION('-', 1), FST::RELATION('+', 1),
FST::RELATION('*', 1), FST::RELATION('/', 1),\
               FST::RELATION('=', 1), FST::RELATION(':', 1),
FST::RELATION(')', 1), FST::RELATION('(', 1),\
               FST::RELATION('}', 1), FST::RELATION('{', 1),
FST::RELATION(']', 1), FST::RELATION('[', 1),\
               FST::RELATION('!', 1), FST::RELATION('?', 1),
FST::RELATION('#', 1), FST::RELATION('&', 1),\
               FST::RELATION('>', 1), FST::RELATION('<', 1),
FST::RELATION('[', 1), FST::RELATION(']', 1), FST::RELATION('%', 1),\
               FST::RELATION('\\', 1), FST::RELATION('|', 1),
FST::RELATION('~', 1), FST::RELATION('\'', 2)), \
     FST::NODE()

#define GRAPH_INT_LITERAL 3, \
     FST::NODE(11, \
               FST::RELATION('-', 1),\
               FST::RELATION('0',2),FST::RELATION('1',2),\
               FST::RELATION('2',2),FST::RELATION('3',2),\
               FST::RELATION('4',2),FST::RELATION('5',2),\
               FST::RELATION('6',2),FST::RELATION('7',2),\
               FST::RELATION('8',2),FST::RELATION('9',2)),\
     FST::NODE(9,\
               FST::RELATION('1',2),\
               FST::RELATION('2',2),FST::RELATION('3',2),\
               FST::RELATION('4',2),FST::RELATION('5',2),\
               FST::RELATION('6',2),FST::RELATION('7',2),\
               FST::RELATION('8',2),FST::RELATION('9',2)),\
     FST::NODE(10,\
               FST::RELATION('0',2),FST::RELATION('1',2),\
               FST::RELATION('2',2),FST::RELATION('3',2),\
               FST::RELATION('4',2),FST::RELATION('5',2),\
               FST::RELATION('6',2),FST::RELATION('7',2),\
               FST::RELATION('8',2),FST::RELATION('9',2))

#define GRAPH_TRUE 5, \
     FST::NODE(1, FST::RELATION('t',1)),\
     FST::NODE(1, FST::RELATION('r',2)),\
     FST::NODE(1, FST::RELATION('u',3)),\
     FST::NODE(1, FST::RELATION('e',4)),\
     FST::NODE()
```

```
#define GRAPH_FALSE 6, \
     FST::NODE(1, FST::RELATION('f',1)),\
     FST::NODE(1, FST::RELATION('a',2)),\
     FST::NODE(1, FST::RELATION('l',3)),\
     FST::NODE(1, FST::RELATION('s',4)),\
     FST::NODE(1, FST::RELATION('e',5)),\
     FST::NODE()

#define GRAPH_FUNCTION 9, \
     FST::NODE(1, FST::RELATION('f', 1)),\
    FST::NODE(1, FST::RELATION('u', 2)),\
    FST::NODE(1, FST::RELATION('n', 3)),\
    FST::NODE(1, FST::RELATION('c', 4)),\
    FST::NODE(1, FST::RELATION('t', 5)),\
    FST::NODE(1, FST::RELATION('i', 6)),\
    FST::NODE(1, FST::RELATION('o', 7)),\
    FST::NODE(1, FST::RELATION('n', 8)),\
    FST::NODE()

#define GRAPH_NUMBER 8,\
     FST::NODE(1, FST::RELATION('i',1)),\
     FST::NODE(1, FST::RELATION('n',2)),\
     FST::NODE(1, FST::RELATION('t',3)),\
     FST::NODE(1, FST::RELATION('e',4)),\
     FST::NODE(1, FST::RELATION('g',5)),\
     FST::NODE(1, FST::RELATION('e',6)),\
     FST::NODE(1, FST::RELATION('r',7)),\
     FST::NODE()

#define GRAPH_UNSIGNED 9,\
     FST::NODE(1, FST::RELATION('u',1)),\
     FST::NODE(1, FST::RELATION('n',2)),\
     FST::NODE(1, FST::RELATION('s',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('g',5)),\
     FST::NODE(1, FST::RELATION('n',6)),\
     FST::NODE(1, FST::RELATION('e',7)),\
     FST::NODE(1, FST::RELATION('d',8)),\
     FST::NODE()

#define GRAPH_BOOL 5, \
     FST::NODE(1, FST::RELATION('b',1)),\
     FST::NODE(1, FST::RELATION('o',2)),\
     FST::NODE(1, FST::RELATION('o',3)),\
     FST::NODE(1, FST::RELATION('l',4)),\
     FST::NODE()

#define GRAPH_STRING 7, \
     FST::NODE(1, FST::RELATION('s',1)),\
     FST::NODE(1, FST::RELATION('t',2)),\
     FST::NODE(1, FST::RELATION('r',3)),\
     FST::NODE(1, FST::RELATION('i',4)),\
     FST::NODE(1, FST::RELATION('n',5)),\
```

```
     FST::NODE(1, FST::RELATION('g',6)),\
     FST::NODE()

#define GRAPH_MAIN 5, \
     FST::NODE(1,FST::RELATION('m',1)),\
     FST::NODE(1,FST::RELATION('a',2)),\
     FST::NODE(1,FST::RELATION('i',3)),\
     FST::NODE(1,FST::RELATION('n',4)),\
     FST::NODE()

#define GRAPH_DO 3, \
     FST::NODE(1,FST::RELATION('d',1)),\
     FST::NODE(1,FST::RELATION('o',2)),\
     FST::NODE()

#define GRAPH_WHILE 6, \
     FST::NODE(1,FST::RELATION('w',1)),\
     FST::NODE(1,FST::RELATION('h',2)),\
     FST::NODE(1,FST::RELATION('i',3)),\
     FST::NODE(1,FST::RELATION('l',4)),\
     FST::NODE(1,FST::RELATION('e',5)),\
     FST::NODE()

#define GRAPH_IF 3, \
    FST::NODE(1, FST::RELATION('i',1)), \
    FST::NODE(1, FST::RELATION('f',2)), \
    FST::NODE()

#define GRAPH_COUT 5, \
     FST::NODE(1, FST::RELATION('c',1)),\
     FST::NODE(1, FST::RELATION('o',2)),\
     FST::NODE(1, FST::RELATION('u',3)),\
     FST::NODE(1, FST::RELATION('t',4)),\
     FST::NODE()

#define GRAPH_RETURN 7, \
     FST::NODE(1, FST::RELATION('r',1)),\
     FST::NODE(1, FST::RELATION('e',2)),\
     FST::NODE(1, FST::RELATION('t',3)),\
     FST::NODE(1, FST::RELATION('u',4)),\
     FST::NODE(1, FST::RELATION('r',5)),\
     FST::NODE(1, FST::RELATION('n',6)),\
     FST::NODE()

#define GRAPH_VOID 5, \
     FST::NODE(1, FST::RELATION('v',1)),\
     FST::NODE(1, FST::RELATION('o',2)),\
     FST::NODE(1, FST::RELATION('i',3)),\
     FST::NODE(1, FST::RELATION('d',4)),\
     FST::NODE()

#define GRAPH_DECLARE 8, \
     FST::NODE(1,FST::RELATION('d',1)),\
```

```
     FST::NODE(1,FST::RELATION('e',2)),\
     FST::NODE(1,FST::RELATION('c',3)),\
     FST::NODE(1,FST::RELATION('l',4)),\
     FST::NODE(1,FST::RELATION('a',5)),\
     FST::NODE(1,FST::RELATION('r',6)),\
     FST::NODE(1,FST::RELATION('e',7)),\
     FST::NODE()

#define GRAPH_HEX_LITERAL 4, \
    FST::NODE(2, FST::RELATION('-', 0), FST::RELATION('0', 1)), \
    FST::NODE(1, FST::RELATION('x', 2)), \
    FST::NODE(44, \
          FST::RELATION('0', 2), FST::RELATION('1', 2),
FST::RELATION('2', 2), FST::RELATION('3', 2), \
        FST::RELATION('4', 2), FST::RELATION('5', 2),
FST::RELATION('6', 2), FST::RELATION('7', 2), \
        FST::RELATION('8', 2), FST::RELATION('9', 2),
FST::RELATION('A', 2), FST::RELATION('B', 2), \
        FST::RELATION('C', 2), FST::RELATION('D', 2),
FST::RELATION('E', 2), FST::RELATION('F', 2), \
        FST::RELATION('a', 2), FST::RELATION('b', 2),
FST::RELATION('c', 2), FST::RELATION('d', 2), \
        FST::RELATION('e', 2), FST::RELATION('f', 2) ,\
        FST::RELATION('0', 3), FST::RELATION('1', 3),
FST::RELATION('2', 3), FST::RELATION('3', 3), \
        FST::RELATION('4', 3), FST::RELATION('5', 3),
FST::RELATION('6', 3), FST::RELATION('7', 3), \
        FST::RELATION('8', 3), FST::RELATION('9', 3),
FST::RELATION('A', 3), FST::RELATION('B', 3), \
        FST::RELATION('C', 3), FST::RELATION('D', 3),
FST::RELATION('E', 3), FST::RELATION('F', 3), \
        FST::RELATION('a', 3), FST::RELATION('b', 3),
FST::RELATION('c', 3), FST::RELATION('d', 3), \
        FST::RELATION('e', 3), FST::RELATION('f', 3) \
    ), \
    FST::NODE()
          ));
```

Листинг 1 – Грамматика языка RIA-2025

```
     struct MfstState                    //состояние автомата(для
сохранения
{
     short lenta_position;          //позиция на ленте
     short nrule;                   //номер текущего правила
     short nrulechain;              //номер текущей цепочки
     MFSTSTSTACK st;                     //стек автомата
     MfstState();
     MfstState(
          short pposition,          //позиция на ленте
          MFSTSTSTACK pst,          //стек автомата
          short pnrulechain         //номер текущей цепочки,
текущего правила
     );
```

```
     MfstState(
          short pposition,         //позиция на ленте
          MFSTSTSTACK pst,         //стек автомата
          short pnrule,            //номер текущего правила
          short pnrulechain        //номер текущей цепочки,
текущего правила
     );
};

struct Mfst                         //магазинный автомат
{
     enum RC_STEP             //шаг автомата
     {
          NS_OK,                   //найдено правило и цепочка,
цепочка записана в стек
          NS_NORULE,               //не найдено правило
грамматики (ошибки в грамматике)
          NS_NORULECHAIN,          //не найдена подходящая
цепочка правила (ошибка в исходном коде)
          NS_ERROR,           //неизвестный нетерминальный символ
грамматики
          TS_OK,                   //текущий символ ленты ==
вершине стека, продвинулась лента, pop стека
          TS_NOK,                  //текущий символ ленты !=
вершине стека, восстановлено состояние
          LENTA_END,               //текущая позиция ленты >=
lenta_size
          SURPRISE            //неожиданный код возврата ( ошибка
в step)
     };
     struct MfstDiagnosis          //диагностика
     {
          short lenta_position;    //позиция на ленте
          RC_STEP rc_step;         //код завершения шага
          short nrule;             //номер правила
          short nrule_chain;       //номер цепочки правила
          MfstDiagnosis();
          MfstDiagnosis(           //диагностика
               short plenta_position,   //позиция на ленте
               RC_STEP prc_step,        //код завершения шага
               short pnrule,            //номер правила
               short pnrule_chain       //номер цепочки правила
          );
     }
     diagnosis[MFST_DIAGN_digit];  //последние самые глубокие
сообщения

     class my_stack_MfstState :public std::stack<MfstState> {
     public:
          using std::stack<MfstState>::c;
     };

     GRBALPHABET* lenta;               //перекодированная(TS/NS)
```

```
лента (из LEX)
     short lenta_position;              //текущая позиция на
ленте
     short nrule;                       //номер текущего правила
     short nrulechain;                  //номер текущей цепочки,
текущего правила
     short lenta_size;                  //размер ленты
     GRB::Greibach grebach;             //грамматика Грейбах
     Lexer::LEX lex;                         //результат работы
лексического анализатора
     MFSTSTSTACK st;                         //стек автомата

     my_stack_MfstState storestate;     //стек для сохранения
состояний
     Mfst();
     Mfst(
          Lexer::LEX plex,              //результат работы
лексического анализатора
          GRB::Greibach pgrebach        //грамматика Грейбах);
     char* getCSt(char* buf);           //получить содержимое
стека
     char* getCLenta(char* buf, short pos, short n = 25);//лента:
n символов с pos
     char* getDiagnosis(short n, char* buf);          //получить
n-ую строку диагностики или 0x00
     bool savestate(const Log::LOG& log);
     //сохранить состояние автомата
     bool reststate(const Log::LOG& log);
     //восстановить состояние автомата
     bool push_chain(                        //поместить уепочку
правила в стек
          GRB::Rule::Chain chain                 //цепочка
правила);
     RC_STEP step(const Log::LOG& log);          //выполнить шаг
автомата
     bool start(const Log::LOG& log);            //запустить
автомат
     bool savediagnois(
          RC_STEP pprc_step             //код завершения шага);
     void printrules(const Log::LOG& log);       //вывести
последовательность правил
     struct Deducation        //вывод
     {
          short size;         //количество шагов в выводе
          short* nrules; //номера правил грамматики
          short* nrulechains;//номера цепочек правил грамматики
(nrules)
          Deducation() { size = 0; nrules = 0; nrulechains =
0; };} deducation;
     bool savededucation();   //сохранить дерево вывода};
};
```

Листинг 2 – Структура магазинного автомата

```
struct Greibach                //грамматика Грейбах
{
     short size;               //количество правил
     GRBALPHABET startN; //стартовый символ
     GRBALPHABET stbottomT;//дно стека
     Rule* rules;              //множество правил
     Greibach() { short size = 0; startN = 0; stbottomT = 0;
rules = 0; };
     Greibach(
          GRBALPHABET pstartN,          //стартовый символ
          GRBALPHABET pstbootomT,       //дно стека
          short psize,                  //количество правил
          Rule r, ...                        //правила
     );
     short getRule(      //получить правило, возвращается номер
правила или -1
          GRBALPHABET pnn,    //левый символ правила
          Rule& prule              //возвращаемое правило
грамматики
     );
     Rule getRule(short n);   //получить правило по номеру
};};
```

Листинг 3 – Структура грамматики Грейбах

```
0    :S->fti(P){BS         fti(ti,ti){vti;i=i+i;ei;}      S$
1    : SAVESTATE:           1
1    :                     fti(ti,ti){vti;i=i+i;ei;}
fti(P){BS$
2    :                     ti(ti,ti){vti;i=i+i;ei;}f      ti(P){BS$
3    :                     i(ti,ti){vti;i=i+i;ei;}ft      i(P){BS$
4    :                     (ti,ti){vti;i=i+i;ei;}fti      (P){BS$
5    :                     ti,ti){vti;i=i+i;ei;}fti(      P){BS$
6    :P->ti                ti,ti){vti;i=i+i;ei;}fti(      P){BS$
7    : SAVESTATE:           2
7    :                     ti,ti){vti;i=i+i;ei;}fti(      ti){BS$
8    :                     i,ti){vti;i=i+i;ei;}fti(t      i){BS$
9    :                     ,ti){vti;i=i+i;ei;}fti(ti      ){BS$
....
1792: TNS_NORULECHAIN/NS_NORULE
1792: RESSTATE
1792:                      l;}                            E;B$
1793:E->l                  l;}                            E;B$
1794: SAVESTATE:           225
1794:                      l;}                            l;B$
1795:                      ;}                             ;B$
1796:                      }                              B$
1797:B->}                  }                              B$
1798: SAVESTATE:           226
```

```
1798:                       }                                   }$
1799:                                                           $
1800: LENTA_END
1801: ------>LENTA_END
```

Листинг 4 - Разбор исходного кода синтаксическим анализатором

```
0    : S->fti(P){BS
4    : P->ti,P
7    : P->ti
11   : B->vti;B
15   : B->i=E;B
17   : E->iZE
18   : Z->+
19   : E->i
21   : B->eE;B
22   : E->i
24   : B->}
25   : S->fti(P){BS
29   : P->ti
33   : B->eE;B
34   : E->(E)ZE
35   : E->iZE
36   : Z->%
37   : E->l
39   : Z->E
40   : E->l
…
375  : E->l
377  : B->vti;B
381  : B->i=E;B
383  : E->iZE
384  : Z->|
385  : E->i
387  : B->oE;B
388  : E->i
390  : B->oE;B
391  : E->l
393  : B->vti;B
397  : B->i=E;B
399  : E->l
401  : B->oE;B
402  : E->l
404  : B->}
```

Листинг 5 – Дерево разбора

Приложение Г

```
int getPriority(LT::Entry& e)
{
     switch (e.lexema)
     {
     case LEX_LEFTHESIS: case LEX_RIGHTTHESIS: return 0;
     case LEX_EQUALS: case LEX_NOTEQUALS: case LEX_MORE: case
LEX_LESS: case LEX_MOREEQUALS: case LEX_LESSEQUALS: return 1;
     case LEX_BITOR: return 2;
     case LEX_BITAND: return 3;
     case LEX_PLUS: case LEX_MINUS: return 4;
     case LEX_STAR: case LEX_DIRSLASH: case LEX_PERSENT: return
5;
     case LEX_BITNOT: return 6;
     case LEX_NOT: return 7;
     default: return -1;
     }
}
bool PolishNotation(Lexer::LEX& tbls, Log::LOG& log)
{
    unsigned curExprBegin = 0;
    ltvec v;
    LT::LexTable new_table = LT::Create(tbls.lextable.maxsize);
    intvec vpositions = getExprPositions(tbls);
    for (int i = 0; i < tbls.lextable.size; i++)
    {
        if (curExprBegin < vpositions.size() && i ==
vpositions[curExprBegin])
        {
            // Determine end of expression
            // For assignment/cout: until LEX_SEPARATOR
            // For if/while: until LEX_RIGHTTHESIS (matching)
            // fillVector needs to be smart or we rely on
vpositions having distinct ranges?
            // getExprPositions returns start indices.
            // We need to know where it ends.

            // Let's look at fillVector. It goes until
LEX_SEPARATOR.
            // This is bad for if(expr).
            // I need to refactor fillVector to take end
position or handle parentheses.

            // Assume getExprPositions returns start of
expression.
            // We need to determine end dynamically.

            int lexcount = 0;

            // Check context
            // If previous was =, cout, return: read until ;
            // If previous was (, and before that if/while: read
```

```
until )

            // Simplified approach:
            // If we are here, we are at the start of an
expression (e.g. after =, or after ( for if)
            // Actually getExprPositions returns index of the
*first token of the expression*.

            // Logic to find end:
            int j = i;
            int balance = 0;
            bool isParenExpr = false;

            // Check if this expression is inside parens
(if/while)
            // Look behind is hard here as we iterate.
            // But we can check if the *terminator* is ; or )

            // Better: fillVectorUntilTerminator

            v.clear();
            for (; j < tbls.lextable.size; j++) {
                if (tbls.lextable.table[j].lexema ==
LEX_SEPARATOR) {
                     if (balance == 0) break;
                }
                if (tbls.lextable.table[j].lexema ==
LEX_LEFTHESIS) balance++;
                if (tbls.lextable.table[j].lexema ==
LEX_RIGHTTHESIS) {
                    if (balance == 0) {
                        // Found closing parenthesis of if/while
condition?
                        // Or just end of (a+b)?
                        // If we started inside ( e.g. if ( a...
                        // We rely on getExprPositions logic.
                        // If getExprPositions pushed index of
'a', then we scan until ')'
                        // But how to distinguish 'a + (b)' from
'if (a) { ... }'?
                        // We need to stop at the ')' that
closes the condition.
                        // If we assume expressions in if/while
are simple, we stop at first unbalanced )
                         break;
                    }
                    balance--;
                }
                 if (tbls.lextable.table[j].lexema == LEX_LEFT)
break; // Safety break
                 v.push_back(LT::Entry(tbls.lextable.table[j]));
            }
            lexcount = v.size();
```

```
            if (lexcount > 0) // Changed from > 1 to 0 (single
literal is expr)
            {
                bool rc = setPolishNotation(tbls.idtable, log,
vpositions[curExprBegin], v);
                if (!rc)
                    return false;
            }

            addToTable(new_table, tbls.idtable, v);
            i += lexcount - 1; // Skip processed tokens
            curExprBegin++;
            continue;
        }

        if (tbls.lextable.table[i].lexema == LEX_ID ||
tbls.lextable.table[i].lexema == LEX_LITERAL ||
tbls.lextable.table[i].lexema == LEX_TRUE ||
tbls.lextable.table[i].lexema == LEX_FALSE)
        {
            int firstind = Lexer::getIndexInLT(new_table,
tbls.lextable.table[i].idxTI);
            if (firstind == -1)
                firstind = new_table.size;
            if (tbls.lextable.table[i].idxTI != NULLIDX_TI)

tbls.idtable.table[tbls.lextable.table[i].idxTI].idxfirstLE =
firstind;
        }
        LT::Add(new_table, tbls.lextable.table[i]);
    }

    tbls.lextable = new_table;
    return true;
}

// fillVector removed/inlined above

void addToTable(LT::LexTable& new_table, IT::IdTable& idtable,
ltvec& v)
{
    for (unsigned i = 0; i < v.size(); i++)
    {
        LT::Add(new_table, v[i]);
        if (v[i].lexema == LEX_ID || v[i].lexema == LEX_LITERAL)
        {
            if (v[i].idxTI != NULLIDX_TI) {
                int firstind = Lexer::getIndexInLT(new_table,
v[i].idxTI);
                idtable.table[v[i].idxTI].idxfirstLE = firstind;
            }
        }
```

```
    }
}

intvec getExprPositions(Lexer::LEX& tbls)
{
    intvec v;
    for (int i = 0; i < tbls.lextable.size; i++)
    {
        // Case 1: Assignment: ID = Expr ;
        if (tbls.lextable.table[i].lexema == LEX_EQUAL)
        {
            v.push_back(i + 1);
        }
        // Case 2: Return: return Expr ;
        else if (tbls.lextable.table[i].lexema == LEX_RETURN)
        {
            v.push_back(i + 1);
        }
        // Case 3: cout Expr ;
        else if (tbls.lextable.table[i].lexema == LEX_COUT)
        {
            v.push_back(i + 1);
        }
        // Case 4: if ( Expr )
        else if (tbls.lextable.table[i].lexema == LEX_LEFTHESIS)
        {
            if (i > 0 && (tbls.lextable.table[i-1].lexema ==
LEX_WHILE || tbls.lextable.table[i-1].lexema == LEX_IF))
            {
                 v.push_back(i + 1);
            }
        }
    }
    return v;
}

bool __cdecl setPolishNotation(IT::IdTable& idtable, Log::LOG&
log, int lextable_pos, ltvec& v)
{
    vector<LT::Entry> result;
    stack<LT::Entry> s;
    bool ignore = false;

    for (unsigned i = 0; i < v.size(); i++)
    {
        if (ignore)
        {
            result.push_back(v[i]);
            if (v[i].lexema == LEX_RIGHTTHESIS)
                ignore = false;
            continue;
        }
```

```
        // Handle function calls within expression (simple skip)
        if (v[i].lexema == LEX_ID && v.size() > i+1 &&
v[i+1].lexema == LEX_LEFTHESIS) {
            // It's a function call like func(a,b)
            // We treat it as a single operand for PN purposes
(simplified)
            // But PN logic separates operands.
            // Current logic ignores content of function call
params?
            // "idtable... idtype == IT::IDTYPE::F" checks if ID
is a function.
            // We rely on that.
        }

        int priority = getPriority(v[i]);

        if (priority != -1 || v[i].lexema == LEX_LEFTHESIS ||
v[i].lexema == LEX_RIGHTTHESIS)
        {
            if (s.empty() || v[i].lexema == LEX_LEFTHESIS)
            {
                s.push(v[i]);
                continue;
            }

            if (v[i].lexema == LEX_RIGHTTHESIS)
            {
                while (!s.empty() && s.top().lexema !=
LEX_LEFTHESIS)
                {
                    result.push_back(s.top());
                    s.pop();
                }
                if (!s.empty() && s.top().lexema ==
LEX_LEFTHESIS)
                    s.pop();
                continue;
            }

            while (!s.empty() && getPriority(s.top()) >=
priority)
            {
                result.push_back(s.top());
                s.pop();
            }
            s.push(v[i]);
        }
        else
        {
             // Operand (ID, Literal)
             if (v[i].idxTI != NULLIDX_TI &&
(idtable.table[v[i].idxTI].idtype == IT::IDTYPE::F ||
idtable.table[v[i].idxTI].idtype == IT::IDTYPE::S))
```

```
                ignore = true; // Skip function params in PN
            result.push_back(v[i]);
        }
    }

    while (!s.empty()) { result.push_back(s.top()); s.pop(); }
    v = result;
    return true;
}
```

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

```
1   | fti[0](ti[1],ti[2]){
2   | vti[3];
3   | i[3]=i[1]i[2]+;
4   | ei[3];
5   | }
7   | fti[4](ti[5]){
8   | ei[5]l[6]%l[7]E;
9   | }
13  | m{
14  | vti[8];
15  | vti[9];
16  | vti[10];
17  | vti[11];
18  | vti[12];
19  | vti[13];
21  | ol[14];
23  | i[8]=l[15];
24  | i[9]=l[16];
25  | ol[17];
26  | ol[18];oi[0](i[8],i[9]);ol[19];
27  | ol[20];oi[8]i[9]-;ol[21];
28  | ol[22];oi[8]i[9]*;ol[23];
29  | ol[24];oi[8]i[9]/;ol[25];
30  | ol[26];oi[8]i[9]%;ol[27];
32  | ol[28];oi[8]i[9]|;ol[29];
33  | ol[30];oi[8]i[9]&;ol[31];
34  | vti[32];
35  | i[32]=l[16];
36  | i[13]=i[32]~;
37  | ol[33];oi[13];ol[34];
39  | ol[35];
40  | i[12]=l[36];
41  | ?(i[12]!){
42  | ol[37];
43  | }
44  | ?(i[8]i[9]N){
45  | ol[38];
46  | }
49  | ol[39];
50  | ol[40];
```

```
51  | ?(i[4](l[41])){
52  | ol[42];
53  | }
55  | vti[43];
58  | ol[44];
59  | i[11]=l[45];
60  | ol[46];oi[11];ol[47];
61  | ol[48];oi[49](i[11]);ol[50];
62  | ol[51];oi[52](l[53]);ol[54];
64  | vti[55];
65  | i[55]=i[52](i[8]);
66  | oi[55];ol[56];
68  | ol[57];
69  | i[8]=l[7];
70  | d{
71  | ol[58];oi[8];ol[59];
72  | i[8]=i[8]l[60]+;
73  | }w(i[8]l[16]N);
75  | vti[61];
76  | i[61]=l[7];
77  | ol[6]i[61]+;ol[62];
79  | vti[63];
80  | vti[64];
81  | i[63]=l[15];
82  | i[64]=l[65];
83  | vti[66];
84  | i[66]=i[63]i[64]l[6]/+;
85  | oi[66];ol[67];
87  | vti[68];
88  | i[68]=i[63]i[64]|;
89  | oi[68];ol[69];
91  | vti[70];
92  | i[70]=l[71];
94  | ol[72];
99  | }
```

Листинг 6 – результат преобразования к обратной польской записи

Приложение Д

```
.586
.model flat, stdcall
includelib legacy_stdio_definitions.lib
includelib msvcrtd.lib
includelib vcruntimed.lib
includelib ucrtd.lib
includelib kernel32.lib
includelib "..\RIA-2025\Debug\StaticLibrary.lib"
ExitProcess PROTO:DWORD
.stack 4096 outlich PROTO : DWORD
 outrad PROTO : DWORD
 str_len PROTO : DWORD, : DWORD
 strlen EQU str_len
 tostring PROTO : DWORD, : DWORD
 atoii  PROTO : DWORD,  : DWORD
.const
          newline byte 13, 10, 0
          var_LTRL1 sdword 2
          var_LTRL2 sdword 0
          var_LTRL3 byte 'StartDemo', 0
          var_LTRL4 sdword 10
          var_LTRL5 sdword 3
          var_LTRL6 byte 'x = 10, y = 3', 10, 0
          var_LTRL7 byte 'x + y = ', 0
          var_LTRL8 byte 10, 0
          var_LTRL9 byte 'x - y = ', 0
          var_LTRL10 byte 10, 0
          var_LTRL11 byte 'x * y = ', 0
          var_LTRL12 byte 10, 0
          var_LTRL13 byte 'x / y = ', 0
          var_LTRL14 byte 10, 0
          var_LTRL15 byte 'x % y = ', 0
          var_LTRL16 byte 10, 0
          var_LTRL17 byte 'Bitwise OR (x | y): ', 0
          var_LTRL18 byte 10, 0
          var_LTRL19 byte 'Bitwise AND (x & y): ', 0
          var_LTRL20 byte 10, 0
          var_LTRL21 byte 'Bitwise NOT (~vbn): ', 0
          var_LTRL22 byte 10, 0
          var_LTRL23 byte 'Logic & If:', 10, 0
          var_LTRL24 sdword 0
          var_LTRL25 byte 'Flag is true.', 10, 0
          var_LTRL26 byte 'x is not equal to y', 10, 0
          var_LTRL27 byte 'Function & Bool:', 10, 0
          var_LTRL28 byte 'Check if 4 is even: ', 0
          var_LTRL29 sdword 4
          var_LTRL30 byte 'Yes', 10, 0
          var_LTRL31 byte 'Strings:', 10, 0
          var_LTRL32 byte 'Hello', 9, 'World', 0
          var_LTRL33 byte 'String with tab: ', 0
          var_LTRL34 byte 10, 0
```

```
          var_LTRL35 byte 'Length of string: ', 0
          var_LTRL36 byte 10, 0
          var_LTRL37 byte 'Int to String: ', 0
          var_LTRL38 sdword 12345
          var_LTRL39 byte 10, 0
          var_LTRL40 byte 10, 0
          var_LTRL41 byte 'Do-While Loop:', 10, 0
          var_LTRL42 byte 'Iter: ', 0
          var_LTRL43 byte 10, 0
          var_LTRL44 sdword 1
          var_LTRL45 byte 10, 0
          var_LTRL46 sdword 40
          var_LTRL47 byte 10, 0
          var_LTRL48 byte 10, 0
          var_LTRL49 sdword 250
          var_LTRL50 byte 'End of Demo', 10, 0.data
          temp sdword ?
          buffer byte 256 dup(0)
          var_sumres dword 0
          var_x dword 0
          var_y dword 0
          var_varname byte 0
          var_s dword 0
          var_flag dword 0
          var_not_vbn byte 0
          var_vbn byte 0
          var_text dword 0
          var_ss dword 0
          var_e byte 0
          var_o byte 0
          var_b byte 0
          var_u byte 0
          var_bit_or byte 0
          var_overflow_test byte 0.code
;----------- sum ------------
sum PROC,
     var_suma : dword, var_sumb : dword
; --- save registers ---
push ebx
push edx
; ----------------------
push var_suma
push var_sumb
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov var_sumres, eax

push var_sumres
pop eax
; --- restore registers ---
```

```
pop edx
pop ebx
; -------------------------
ret
sum ENDP
;-----------------------------
;----------- isEven ------------
isEven PROC,
     var_isEvenv : dword
; --- save registers ---
push ebx
push edx
; ----------------------
push var_isEvenv
push var_LTRL1
pop ebx
pop eax
cdq
mov edx,0
idiv ebx
push edx
push var_LTRL2
pop ebx
pop eax
cmp eax, ebx
sete al
movzx eax, al
push eax
pop eax
; --- restore registers ---
pop edx
pop ebx
; -------------------------
ret
isEven ENDP
;-----------------------------
;----------- MAIN ------------
main PROC
push offset var_LTRL3
call outrad
push var_LTRL4
pop eax
mov var_x, eax
push var_LTRL5
pop eax
mov var_y, eax
push offset var_LTRL6
call outrad
push offset var_LTRL7
call outrad
push var_y
push var_x
call sum
```

```
push eax
call outlich
push offset var_LTRL8
call outrad
push offset var_LTRL9
call outrad
push var_x
push var_y
pop ebx
pop eax
sub eax, ebx
push eax
call outlich
push offset var_LTRL10
call outrad
push offset var_LTRL11
call outrad
push var_x
push var_y
pop ebx
pop eax
imul eax, ebx
push eax
call outlich
push offset var_LTRL12
call outrad
push offset var_LTRL13
call outrad
push var_x
push var_y
pop ebx
pop eax
cdq
idiv ebx
push eax
call outlich
push offset var_LTRL14
call outrad
push offset var_LTRL15
call outrad
push var_x
push var_y
pop ebx
pop eax
cdq
mov edx,0
idiv ebx
push edx
call outlich
push offset var_LTRL16
call outrad
push offset var_LTRL17
call outrad
```

```
push var_x
push var_y
pop ebx
pop eax
or eax, ebx
push eax
call outlich
push offset var_LTRL18
call outrad
push offset var_LTRL19
call outrad
push var_x
push var_y
pop ebx
pop eax
and eax, ebx
push eax
call outlich
push offset var_LTRL20
call outrad
push var_LTRL5
pop eax
mov byte ptr [var_vbn], al
movzx eax, byte ptr [var_vbn]
push eax
pop eax
not eax
push eax
pop eax
mov byte ptr [var_not_vbn], al
push offset var_LTRL21
call outrad
movzx eax, byte ptr [var_not_vbn]
push eax
call outlich
push offset var_LTRL22
call outrad
push offset var_LTRL23
call outrad
push var_LTRL24
pop eax
mov var_flag, eax
push var_flag
pop eax
cmp eax, 0
sete al
movzx eax, al
push eax
pop eax
cmp eax, 0
je lbl_if_end_1
push offset var_LTRL25
call outrad
```

```
lbl_if_end_1:
push var_x
push var_y
pop ebx
pop eax
cmp eax, ebx
setne al
movzx eax, al
push eax
pop eax
cmp eax, 0
je lbl_if_end_2
push offset var_LTRL26
call outrad
lbl_if_end_2:
push offset var_LTRL27
call outrad
push offset var_LTRL28
call outrad
push var_LTRL29
call isEven
push eax
pop eax
cmp eax, 0
je lbl_if_end_3
push offset var_LTRL30
call outrad
lbl_if_end_3:
push offset var_LTRL31
call outrad
push offset var_LTRL32
pop eax
mov var_s, eax
push offset var_LTRL33
call outrad
push var_s
call outrad
push offset var_LTRL34
call outrad
push offset var_LTRL35
call outrad
push var_s
push offset buffer
call strlen
push eax
call outlich
push offset var_LTRL36
call outrad
push offset var_LTRL37
call outrad
push var_LTRL38
push offset buffer
call tostring
```

```
push eax
call outrad
push offset var_LTRL39
call outrad
push var_x
push offset buffer
call tostring
push eax
pop eax
mov var_ss, eax
push var_ss
call outrad
push offset var_LTRL40
call outrad
push offset var_LTRL41
call outrad
push var_LTRL2
pop eax
mov var_x, eax
lbl_do_4:
push offset var_LTRL42
call outrad
push var_x
call outlich
push offset var_LTRL43
call outrad
push var_x
push var_LTRL44
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov var_x, eax
push var_x
push var_LTRL5
pop ebx
pop eax
cmp eax, ebx
setne al
movzx eax, al
push eax
pop eax
cmp eax, 0
jne lbl_do_4
push var_LTRL2
pop eax
mov byte ptr [var_e], al
push var_LTRL1
movzx eax, byte ptr [var_e]
push eax
pop ebx
pop eax
```

```
add eax, ebx
push eax
call outlich
push offset var_LTRL45
call outrad
push var_LTRL4
pop eax
mov byte ptr [var_o], al
push var_LTRL46
pop eax
mov byte ptr [var_b], al
movzx eax, byte ptr [var_o]
push eax
movzx eax, byte ptr [var_b]
push eax
push var_LTRL1
pop ebx
pop eax
cdq
idiv ebx
push eax
pop ebx
pop eax
add eax, ebx
push eax
pop eax
mov byte ptr [var_u], al
movzx eax, byte ptr [var_u]
push eax
call outlich
push offset var_LTRL47
call outrad
movzx eax, byte ptr [var_o]push eax
movzx eax, byte ptr [var_b]
push eax pop ebx pop eax
or eax, ebx
push eax
pop eax
mov byte ptr [var_bit_or], al
movzx eax, byte ptr [var_bit_or]
push eax
call outlich
push offset var_LTRL48
call outrad
push var_LTRL49 pop eax
mov byte ptr [var_overflow_test], al
push offset var_LTRL50
call outrad Ret push 0
call ExitProcess
main ENDP
end main
```

Листинг 1 – Сгенерированный код на языке Assembler

```
StartDemox = 10, y = 3
x + y = 13
x - y = 7
x * y = 30
x / y = 3
x % y = 1
Bitwise OR (x | y): 11
Bitwise AND (x & y): 2
Bitwise NOT (~vbn): 252
Logic & If:
Flag is true.
x is not equal to y
Function & Bool:
Check if 4 is even: Yes
Strings:
String with tab: Hello  World
Length of string: 11
Int to String: 12345
10
Do-While Loop:
Iter: 0
Iter: 1
Iter: 2
2
30
42
End of Demo
```

Листинг 2 – Результат работы файла на языке Assemble